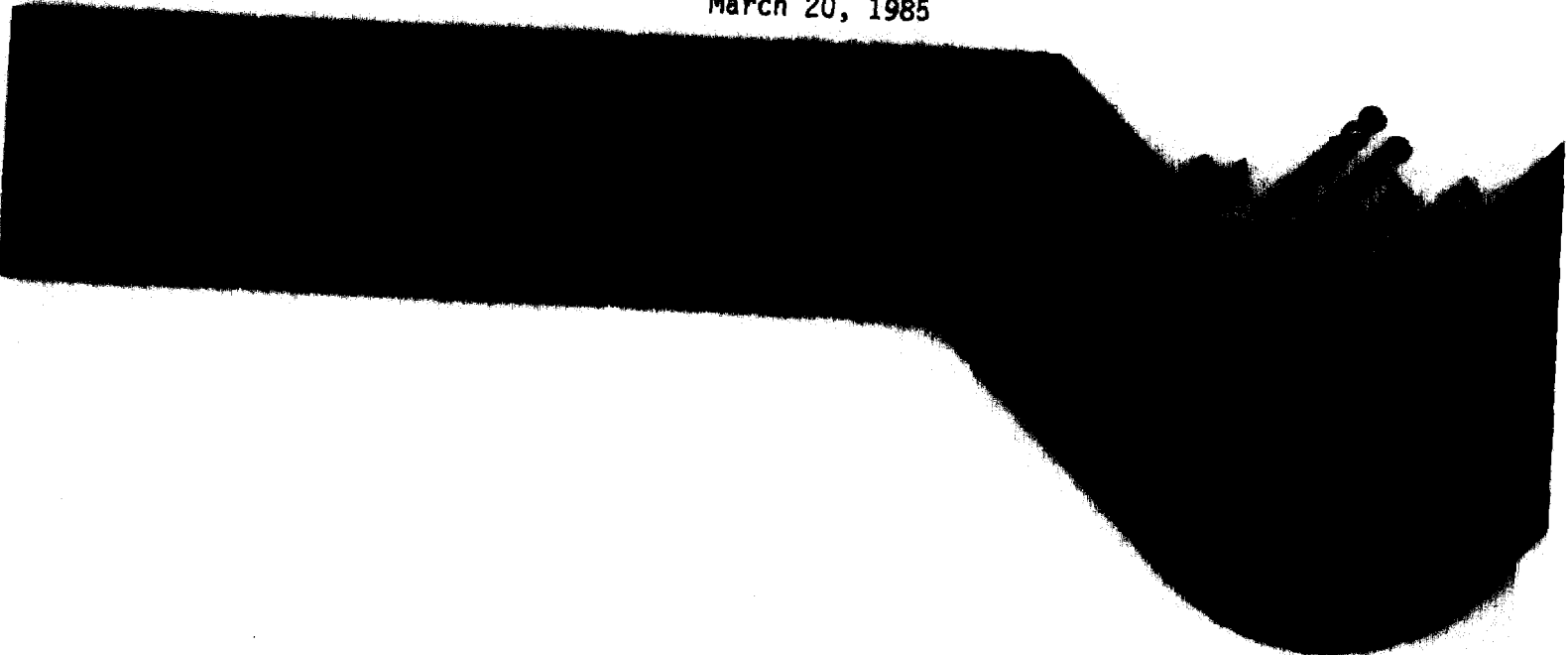

CIRCULATION COPY
SUBJECT TO RECALL
IN TWO WEEKS

UCID- 20378

AMBER KERNEL SPECIFICATION

S-1 Project

March 20, 1985



DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Price: Printed Copy \$; Microfiche \$4.50

<u>Page Range</u>	<u>Domestic Price</u>	<u>Page Range</u>	<u>Domestic Price</u>
001-025	\$ 7.00	326-350	\$ 26.50
026-050	8.50	351-375	28.00
051-075	10.00	376-400	29.50
076-100	11.50	401-426	31.00
101-125	13.00	427-450	32.50
126-150	14.50	451-475	34.00
151-175	16.00	476-500	35.50
176-200	17.50	501-525	37.00
201-225	19.00	526-550	38.50
226-250	20.50	551-575	40.00
251-275	22.00	576-600	41.50
276-300	23.50	601-up ¹	
301-325	25.00		

¹Add 1.50 for each additional 25 page increment, or portion thereof from 601 pages up.

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

S-1 Project

AMBER KERNEL SPECIFICATION

Lawrence Livermore National Laboratory
26 February 1985

PURPOSE

This document provides a detailed definition of the kernel functions provided by the Amber system. The description of each interface includes a discussion of its function, definition of its input and output parameters, and a list of error or warning conditions which result. No attempt is made to provide a tutorial on the possible applications of these interfaces.

Caveat: this is a preliminary document, and the definition of the interfaces is subject to change. It is anticipated that changes will occur more in the form than the function of the interfaces. That is, this document should accurately represent the operations which will ultimately be provided by the Amber kernel, but it is the format and parameterization of the kernel calls that are likely to be modified as the result of ongoing review and experimentation.

In addition, the document does not present a complete set of maintenance functions, such as metering and debugging tools. Appropriate routines will be defined as they are needed.

THE KERNEL AND ITS ROLE IN THE SYSTEM

By design, the Amber system is highly modular with many layers of functionality. Aside from the software engineering advantages, this approach is used to allow particular applications to select the exact set of functions required for their operation, without incurring the overhead of extraneous functions. System support can be tailored for specific applications by configuring the modules or layers which implement the needed operations. Other modules may be simply omitted without additional cost.

The kernel interfaces constitute a layer of functionality. They provide the operations necessary to construct systems for Navy evaluation of the S-1 family of processors. Specifically, support is available for efficient, reliable real-time applications and a sophisticated program development system

The kernel is essentially a high level virtual machine whose operations are more powerful and convenient than those of the raw hardware, and directly reflect the functions in the problem space of intended applications. The kinds of facilities available from the kernel include multitasking on single and multiple processor configurations, control of storage mapping with sharing among different tasks, interprocess communication and synchronization, clock services, a hierarchical file system, input/output for standard devices and support for user-written device drivers, and access control on all objects (files, tasks, devices, etc.).

A concerted attempt has been made to trim the kernel interfaces to the minimum necessary to support the broad range of possible applications. Consequently, the interfaces tend not to be specially suited for any one application, but useful for constructing higher level packages which implement a specialized function. For instance, the kernel provides a single "timer" from which a package supporting multiple timers could be built. A wide variety of such library packages will be provided as part of the Amber Base System. These will reside in the user's address space and execute with the user's level of privilege. Such library routines will be documented elsewhere.

Indeed, the distinction between what is a kernel routine and what is a user-level library routine is of little importance to the user. All can be called directly from high-level language routines, and are documented in the same fashion. The kernel routines merely provide a base from which it is practical to build the more sophisticated user routines. In certain cases, it will be found that a user interface routine provides a specific policy not imposed by the underlying kernel mechanism, or may not utilize all the flexibility available from the kernel. This is done to provide what is considered a simpler or better adapted operation for the applications. When the user interfaces do not provide what is needed, the function can be synthesized directly from the kernel interfaces.

OVERVIEW

A program running on the Amber virtual machine sees a world composed of several classes of objects: domains, tasks, segments and message channels. A program manipulates these objects through the kernel interfaces. For example, a task can be started and stopped, or a segment can be created and cataloged in a domain.

The Storage System

A domain object has two functions. First, it serves as a conventional directory by cataloging other objects including other domains. Second, it serves as a name space by defining the list of objects which can be referenced by a task.

A domain contains "entries" of two kinds. There are the objects that are directly cataloged within the domain, and there are "links" that are synonyms for object entries in the same or other domains. Each entry in a domain has a numeric identifier which is unique across the entire system. This "id" is the name used in most kernel operations. For example, a task identifies an object in a kernel operation by using the id of an entry for the object within the domain in which it is executing. In addition, an entry may have zero or more mnemonic names (of up to 48 characters). These names must be unique only within the domain in which they are defined. A simple lookup operation exists to translate a textual name to an id with which the object can be manipulated. Entries for segment objects may also be referenced by address. By giving a segment entry an address, the user maps it into the address space of the domain. (This is described in detail below.)

The graph of domains, sub-domains and the objects catalogued within them form a strict hierarchical tree. This is called the storage system or domain hierarchy. Any object can be found by giving the ids or names of all entries along the path from the "root" domain to the object's own entry. This route is termed the "pathname" of the object. Of course, an object may also be found by giving the pathname of a link entry that gives the object's actual pathname.

An object can be moved from one domain to another. When this happens, its pathname changes and any links which pointed to the object become invalid. When a domain object is moved, the entire subtree which it heads is, in effect, grafted into another location in the hierarchy. There are no logical restrictions on the movement of objects. However, some restrictions are made on the basis of physical allocation of storage on the long-term storage media (e.g. disks). This is described below. Also, the maximum depth of the hierarchy is limited for implementation reasons.

Status information is kept in the parent domain of each object defined on the system. The kernel itself maintains certain built in attributes such as time of last use, modification or backup, creator, number of times referenced, etc. These are automatically updated as a side effect of kernel operations. User defined attributes are kept on a "property list" similar in function to the Lisp construct of the same name. This is a list of (property name, string value) pairs, which can be updated directly by user programs. The properties are intended to hold information about the use or contents of an object relevant to specific applications.

Access Control

All objects have an "access control list (ACL)" that specifies who can have access to the object. The ACL is a list of pairs (mode set, principal) where the mode set controls the operations which can be performed, and the "principal" stands for the process, user, group, etc. that is to be allowed to perform the operations.

A principal is represented in Amber as a domain object. For instance, to give a task access to some object, one adds to the access control an entry that specifies task's domain of execution as the principal. A (human) user is also represented by a domain specially created to represent his access rights. In order that a task being managed by a user may have the ability to use the access of the user, every domain has an "owner" domain, and the rule is made that a task can utilize access control list entries naming its domain of execution or that domain's owner.

Groups are constructed by a general "power-of-attorney" mechanism. A domain may be granted the right to "use" the access of another. One can then construct a domain which represents a class of users, and control who is a member of that class by altering the access control list. Access granted to the class is then available to its agents.

The kernel does not automatically derive class memberships. To perform some operation on an object, the user must provide an "access-path" for some entry in the ACL. The simplest access-path is a single principal, X, which is either (1) the domain in which the current task is executing, or (2) the owner of this domain. If the principal X does not directly have an entry on the ACL of the object, it may be able to use the power-of-attorney mechanism to get access. For example, if Y appears on the ACL of the object and X has "use" access on the ACL of Y, presentation of the access-path X.Y permits X to obtain Y's access. In general, an access-path is an ordered list of principals X.Y...Z such that the first entry (X) is the current domain or its owner (to which use access is implicit) and in each pair A.B it is the case A has use access to B through its ACL. The access which an access path allows to the object is computed by searching the ACL for an entry naming the last component of the path (Z).

A link to an object contains, in addition to the pathname of the object, an access-path to be used to compute the access on the object. Thus, when a task specifies a link id in a kernel operation, the modes of access are computed from the last component of the access-path. However, no access is allowed unless each component is an agent of the next.

It is important to realize that access is evaluated on each use of a link. Consequently, the modes of access available with a link may vary in time as changes are made to the access control list of the target object. It is possible, for example, to create a link to an object before access is granted to the access-path named in the link. Changes to the access control list of the principal domains listed in the access-path can alter the right to use the access-path as well.

A link also contains a mode field which is voluntary limit on the modes of access to be used; for example, if a segment is only to be read by a task which has read-write access to it, using a link with a read-only mode set prevents inadvertant writes.

When an object (as opposed to a link) is cataloged directly in a task's domain of execution, the id of the object's entry may also be used in a kernel call. In this case, object entry is treated as a link with an access-path consisting of the containing domain. This means that to determine the access rights, the access control list is searched for an entry for the containing domain.

A common case is granting all users on the system access to an object. While this could be accomplished with a kind of super-group, a simplification is permitted. A access control list entry may be created with a special tag denoting the group of all principals; if there is no match for the last component of an access-path, then the universal entry may be used.

Data Storage and Memory Management

Segments are objects that can be used to store data. They are roughly analogous to files on other systems; however, I/O operations are not required to manipulate them. Instead, they may be mapped directly into the virtual memory associated with a domain, and referenced as data by normal instructions. When several programs address the same object, they share identically the same data; changes made by one program are immediately visible to others.

The data in a segment is simply a string of bits. The kernel does not enforce any particular structure on segments; consequently, they may contain any kind of information, in any format that the user requires. (Certain non-kernel functions do utilize segments with special formats, for example, to represent text files and executable code files.)

When a segment is mapped into the virtual memory of a domain, it becomes accessible as the contents of the memory locations over the range of addresses that it occupies. The memory slot occupied by a segment is characterized by two attributes, the base address, and the size. The base address is simply the address of the first bit. The size is the length of the slot. The user must designate the size to be used when mapping the segment. The base address can be either designated by the user or assigned by the kernel. The mapping is not fixed. The segment can be remapped from one location to another and its size changed.

A mapping may be established for each separate segment entry in a domain, and the modes of access allowed to the memory are precisely those that available through the entry. Consequently, if a domain contains more than one entry for a segment, then the segment may appear in its address space in more than one location and with different modes of access.

It is important to remember that a segment can be placed in different locations in a single domain or different domains; thus the contents of a segment should not contain absolute addresses. For this reason, the translators provided to run in applications environments supported by Amber will produce position-independent code.

The architecture forces virtual memory to be allocated in power-of-two multiples of 64K quarter-words. Consequently, the segment slot size is rounded up to the next suitable boundary. The base address must also be aligned on a boundary equal to the rounded size. However, memory use can be controlled in units of pages of 1024 single-words, and regardless of the rounded slot size, the size is enforced with page size granularity.

A segment has a maximum length that defines the largest offset within the segment which can be accessed. This is expressed in terms of quarter-words and enforced to within the limits of the architecture. The maximum length can be increased at will, and decreased to the current length of the segment (the end of the allocated record with the largest offset). There need be no relationship between the maximum or current lengths of a segment and the size allocated to it when it is mapped. If the size is less than the current length, then only a portion of the segment can be addressed. A size greater than the current or maximum lengths allows the segment to be grown.

Physical storage for segments is allocated in units of records (1 page long). When a segment is created, no records of storage are allocated for it until its contents are written. Then a record is allocated for each separate page written. Records which have never been written may be read and appear to be full of zeros. Reading of a zero page does not cause a record to be allocated. Once allocated, a record is not deallocated until the segment is deleted or the segment is explicitly truncated by use of a special operation. Incidental zeroing of a page as a result of normal machine instructions does not cause the corresponding record to be deallocated.

A demand paging mechanism is used to multiplex available system main memory among the pages of addressible segments (those mapped into a domain containing an active task). When a page of a segment is referenced and the page is not already in memory, it is read in from the secondary storage device on which it resides and brought into main memory. The hardware mapping functions place it in the proper location of all referencing domains. Pages which have not been referenced recently are evicted from main memory to make room for incoming pages.

Demand paging algorithms have been shown to be very effective, particularly when the main memory available for paging is large. However, there are certain cases where demand paging has unfortunate side effects. Special escapes are provided to handle these cases. Otherwise, the paging mechanism is completely invisible to the user.

In real-time applications, the possible delays induced by demand paging can prevent a program from meeting response time requirements. These delays are avoided if the storage of a real-time program is kept resident ("wired") in main memory. An operation is provided to allow a privileged program to wire critical segments.

Programs that have long patterns of sequential access, such as large scale vector computations, can result in worse-case performance of page eviction algorithms. (Practical algorithms are approximations of an LRU replacement strategy.) To avoid this, operations are provided which allow the program to indicate to the system its intended pattern of use. Specifically, a program can declare that it is about to reference a section of memory (in a sequential manner), and that it is "done" with a section of memory for the near term.

Certain algorithms, for example in transaction processing, require memory fields to be updated in a particular order so that at each step, the data remains consistent even in the face of a system crash. In our context, this means that the user must be able to control the order in which the secondary storage image of segments is updated. However, the kernel updates records of secondary storage from their in core image in any order it sees fit. A mechanism is therefore provided to allow a program to insure that the main and secondary storage copies of a section of memory are consistent, before proceeding with additional updates.

Secondary Storage Resource Control

Segments are not the only objects that use secondary storage. For example, the storage for domains is handled in precisely the same fashion. (They are in fact segments insofar as the kernel routines are concerned.) Other objects, such as tasks have their "contents" stored in their parent domains. As a result, no storage is allocated specifically for them.

Secondary storage is grouped into "volumes" that may consist of parts of one or more disk-like physical storage devices. When a storage object such as a segment is created, the volume on which its storage is to be placed is

specified. Thereafter, it can be moved from one volume to another if desired. A system of defaults is provided so that the average user need not be explicitly concerned with physical allocation. Each domain has defaults for the volumes on which cataloged objects will be placed.

The kernel distinguishes two types of volumes: first-class and second-class volumes. First-class volumes may hold domains that contain objects residing on other volumes. Second-class volumes may hold domains that point to objects residing only on the same volume. Thus, second-class volumes must contain complete subtrees of the domain hierarchy. (Note that this organization places restriction on the movement of objects from one volume to another, or from one domain to another.) The class distinction exists to allow sections of the hierarchy that can be removed from the system configuration. This may be useful to allow for a graceful degradation of resources in the event of hardware failure, or to achieve better utilization of devices.

There is no operation for defining the class of the volume. It is assumed that this is administrative function performed at the time the storage medium is formatted.

A quota mechanism is provided to control usage of secondary storage. It imposes a limit on the total number of records of a volume that may be used. Quota is applied on a per domain basis, and the limit associated with a domain applies to all segments and subdomains below that domain in the hierarchy. Since a domain may contain objects on different volumes, separate quota controls are provided for each volume.

In order for objects to be created on a volume, a "quota account" must be created to keep track of the storage usage on that volume for objects within the domain. The account encompasses objects in all subdomains that do not themselves have a separate account. Thus, a domain with a quota account forms the root of a "quota subtree"; all objects are charged to the root of the subtree in which they appear. A domain which has a quota account is charged against an account in one of its superiors. Each account has an absolute limit on the number of records which can be used by all objects charged to the account. There need be no relation between the total amount of quota allocated on a volume and the actual number of physical storage records. The creation of a quota account is a privileged system function.

The storage usage on a volume can be further controlled by use of the optional "quota limit"; without an explicit limit, the entire quota allowed by the quota account can be used. The quota limit applies to all subdomains (within a quota subtree), even those which themselves have an explicit quota limit. The usage for each domain is the sum of the usage within that domain and the usages of all subdomains, and the limit at each domain must be satisfied. Setting of a quota limit is a nonprivileged operation, and is intended to be allow a user or group to allocate the resources which they are authorized to use by the presence of the quota account.

The following rules define the boundary conditions, and apply in the case of both the quota account limit and the simple quota limit. In calculating the quota usage, new zero records (records which have never been written into) are not counted, and a program can read from a zero record without affecting the quota usage.

The limit on the use of segment records is absolute. Once the limit has been reached, no additional records can be created. In order to insure correctness of the kernel, domain records must be able to be created arbitrarily. Therefore, domain records are charged against the quota, but are not limited by it. The effect is to reduce the number of records available for segments. However, once the quota has been exceeded, no new objects may be created within the domain, even if they do not take up permanent storage records.

A quota less than the current usage can be set. In such a case, no additional records may be allocated to segments under the domain. If it is desired to use additional records, it is then necessary to destroy other records by deletion of segments or domains, or by truncation of segments.

Task Organization

A task object represents a kind of resource that can execute programs. A primary function of the Amber kernel is to multiplex one or more physical processors among the tasks active on the system. The scheduling mechanism used is a simple priority system. A task may be placed in one of several priority queues, representing the relative acceptable delay in response to events. The scheduler chooses for execution the first runnable task in the queue with the highest priority. When there are several runnable tasks in a priority queue, they are scheduled in a first come first served order. A task is allowed to run for the duration of its runtime quantum; it is then moved to the end of the queue, and its quantum is reset. (A task can, of course, be preempted if a higher priority task becomes runnable.)

There are different sets of queues for each physical processor on the system, and at any one time, a task is assigned to run on only one processor. Explicit action is required to move a task to some other processor. This approach is used for several reasons. First, in a multiprocessor S-1 configuration, external I/O devices are usually accessible from only one of the processors. A task communicating with the device must be restricted to run on that processor. Second, to achieve maximum use of the local caches in the processors, it is important to run a task on the same processor as it was last run. Third, it can serve to achieve concurrency among cooperating tasks. Each of several tasks working on a large problem can be assigned to different processors.

The consequence of this approach is that over the long term the workload assigned to the different processors can become unbalanced. A privileged system task will exist which monitors processor usage by tasks, and redistributes the tasks to different processors. Such action will occur relatively rarely; load balancing thus becomes a "long-term" activity. This system task is not part of the kernel. A single or multiprocessor system could function quite adequately without it. For example, a static assignment of tasks to processors might be sufficient for a well defined real-time application. Being a user-level program, the balancing task can be easily modified or replaced (even dynamically) to tune the system or implement different scheduling policies. For instance, in certain cases load balancing may not be desirable. In an application such as weather forecasting, it may be useful to allocate a small number of processors to interactive control and enquiry tasks, and reserve a larger number of processors to run the large-scale computational tasks.

A task may be in one of several states. An "inactive" task is one which is not allowed to run. When in this state, the machine state of the task is available from the storage system. Once activated, the stored state is considered to be "inconsistent", and should the system crash before the task is deactivated, the state is left inconsistent. A "stopped" task is one whose execution has been momentarily halted. The task machine state remains inconsistent in this state. A "waiting" task is one which has halted its own execution until some event or timeout occurs. A "runable" task is one which is ready to run. The task will be allowed to execute on a processor when it becomes the highest priority, runable task on that processor.

The execution of a task can be controlled by other tasks holding capabilities for it with suitable modes of access. When a task is created, its creator is given full access rights to the task. The task can be activated, started, stopped or deactivated, and its state examined or modified. The state information of a task includes the contents of its registers and program counter. By altering its state, a task can be forced to change the location at which it is executing or to perform a procedure invocation.

A task executes in its parent domain. The entries in the parent domain therefore define the objects which the task can reference. In particular, the task can access as code or data any segments which have been mapped into the virtual memory of the domain. Unlike some other systems, a task in Amber is not the unique possessor of its domain; several tasks may share a domain, thereby retaining close coupling and high bandwidth communication through shared data.

It is the responsibility of the creator of a task to initialize the parent domain so that there exists entries for all segments initially needed by the task (e.g. program, stack, static) appear in the domain, and that the addresses of these segments have been defined. For example, the program segment will be mapped into the domain, and the program counter in the task state set to the start address of the program. Once the task has started execution, it is free to add or delete segments from the domain memory map as it sees fit.

In order that the task have access to modify its domain of execution, it is necessary that a link exist in the domain by which the task can reference the domain. The id of this link is part of the state of the task, and when the "current domain" is used in a kernel operation, this id is used to access the link. The creator of the task is responsible for creating this link.

By these mechanisms, domains can represent the prelinked environment of a task or group of tasks. Often the environment will include certain data segments which themselves must be preinitialized, but which will be modified during execution of the task. The stack and static data segments are examples. When this is the case, it is necessary to create a new domain for each instantiation of the task, and place into the domain private copies of the template objects. As this is expected to be a common operation, a kernel primitive is provided. The user defines a template domain which contains entries for the objects which are required by the task. The template objects are characterized by placing a non-blank "template" property on their property lists. When the kernel is requested to invoke the domain, a new domain is created. Non-template objects are entered into the new domain by creating links to the originals. Copies are made of the template objects and entered into the new domain.

It is expected that the normal mode of operation would be for the copy of a domain to be made an inferior of the domain of the task which invoked it. Such an approach creates a task hierarchy that allows a simple garbage collection of tasks. When the domain of a parent task is deleted, all inferiors are deleted as a consequence of the recursive deletion of domains.

When a domain is "invoked" in this manner, the invoking domain is permitted no more access to the copy of the domain than it had to the template. This allows a domain to invoke a protected domain.

Synchronization

Synchronization between different tasks is accomplished by broadcast mechanism. A task defines a "broadcast event" which signifies some higher level state such as a lock becoming unlocked, a queue becoming empty or alarm clock going off. Whenever a task "broadcasts" the occurrence of the event, all tasks which have defined the event are sent a wakeup.

An event is named by a 2-tuple, object and event-id. That is, for each object in the storage system, there may be an arbitrary number of events differentiated by different ids. The object may be specified by any capability for the object. The id is assigned completely at the discretion of the user program. Consequently, for two tasks to communicate, there need be no pre-registration of the event. The two tasks need only agree on the event-id to be used (or the algorithm for computing it) in advance. For example, a lock event might be defined by the segment containing the lock as the object, and the offset of the lock within the segment as the id.

Special access modes are defined for each object to control the ability to listen for or to broadcast an event. Such control is provided for two reasons: first, to prevent malicious wakeups from being delivered to a task; second, to protect what is, in practice, a channel of communication between two tasks.

Each installation may place a limit on the number of events that may be declared by any single task or by all tasks as a whole.

Note that the system does not explicitly provide queues, semaphores, event counts, locks or other such mechanisms. These can be synthesized using data shared between two communicating tasks (or other mutually observable information) and the event notification mechanism to control the rendezvous between the tasks.

Interprocess Communication

Message channel objects provide a mechanism for communication between tasks. Each channel is bidirectional, and has two "nodes" representing symmetric user and server functions. When a message channel is created, two separate objects are created — one for each node. A message sent from one object is directed to the task listening on the other.

Since access to user and server functions can be passed between domains, several tasks may concurrently have the right to transmit on a channel. However, to avoid conflicts, only one task at a time is allowed to take control of a node. An attempt to take control of a channel that is already in use gives an error indication, though forcible acquisition is allowed. Typically, if a task expects to communicate with more than one other task; one channel is created by which user tasks can request a private channel. Access to the request channel can be resolved by contention algorithms.

Information is transmitted in message packets. A packet is a variable length record, and corresponds to the information transmitted with a single send operation. All packets are delivered to the receiver in the time order in which they were originally sent. The packet mechanism serves three purposes: to buffer the information, to structure the information, and to synchronize transmission.

Data transmission requires a rendezvous between sender and receiver at opposing nodes on a channel. When a message is to be sent, the sender gives the kernel the address and length of the packet. When the receiver asks to read a packet, it is transferred directly from the sender's address space to the receiver's without intermediate buffering. The tasks need not wait at the send or receive operations for the transfer to occur; instead, their execution proceeds asynchronously. Once the transfer actually takes place, the tasks are sent a wakeup indicating that the transaction is complete. A task is allowed to have a single, incomplete send outstanding on each channel. A receiver may have two outstanding reads on each channel, thus providing double buffering.

As a structuring tool, each packet maintains a separate identity, which is visible to the user program. Depending on the application, the program may choose to use or ignore the packet structure of a message. For instance, in a character stream transmission, the packet structure might be incidental. Alternatively, when a message channel is used to transmit requests to a server process, each packet could represent a separate request.

Packet delivery is signalled to the communicating tasks with the wakeup mechanism, thereby providing synchronization and flow control functions. The sender is notified that the packet has been transmitted; the receiver, that the packet has been read. In addition, the receiver can request a wakeup when a packet is available from the sender before issuing a request to read the packet.

Wakeup

The event broadcast mechanism, the message channel mechanism and the timer mechanism use "wakeup" to signal the occurrence of an event to the user program. In addition, a task which has access to another task may send the other task a wakeup directly. Privileged tasks may receive wakeups when an I/O interrupt occurs.

Each task has a small, fixed set of wakeups monitored by the kernel. When a wakeup is sent to the task, it is interrupted. If the task is "blocked" waiting for a wakeup, it is made runnable so that it can process the interrupt; if the task has a higher priority than that which is currently running on the processor, then the task immediately begins to run. If the task is running at the time that the wakeup is received, then its flow of control is altered immediately.

The method used to interrupt a task has not yet been precisely defined. It is intended that software interrupts as defined here follow the general protocol for hardware interrupts, traps, etc. as defined by the S-1 architecture.

A task may mask against all wakeups as a group. Consequently, while a task is masked, or stopped, it may receive any number of wakeups. These are recorded by the kernel in a "pending wakeup mask". When the user task is interrupted, the entire mask is delivered as an interrupt parameter, and the mask is cleared. If two identical wakeups are received before the task can be interrupted, then knowledge that the second wakeup was received is lost.

Each installation may define the size of the wakeup set used by tasks on the system. A normal value would be 576 wakeups per task.

It is anticipated that a library package will be created that provides user programs with a higher level protocols such as waiting for a particular subset of wakeups or interrupt priorities. Note that when an application requires true priority interrupts, it should be implemented as multiple tasks with different scheduler priorities.

Extended Type/Protected Objects

Access to an object can be controlled by a type manager which implements operations or protection policies not directly supported by the system. When it is created, an object may be tagged as being "sealed" by some domain. The result is that normal kernel operations (such as read and write for a segment) can no longer be used to manipulate the object; instead, the type manager (i.e. the sealing domain) must be requested to perform the operation on behalf of the calling program. The type manager (or its agents) may unseal objects sealed by it and thereby use the normal kernel operations to service the request.

The extended type object and its representation are physically distinct objects with separate property lists. Unsealing an extended type object generates a link to the representation; this is the only way in which to reference it.

The modes of access on an extended type object are not interpreted by the kernel (except as noted below). Instead, the modes are left to the interpretation of the extended type manager, which can assign to them entirely new functions.

Certain kernel operations for maintaining the catalog of objects can be directly performed by the user without the intervention of the type manager. In general, extended type objects can be created and deleted, and their property lists examined and modified. The access modes controlling these functions are defined for all objects, including any extended type objects.

Input/Output

I/O services are provided by Amber through three mechanisms: segmentation, direct access to external devices, and interprocess communication.

The functions provided by file systems of other operating systems are subsumed by the segmentation model described above. A file is accessed by mapping a segment into the memory of a domain. The segment can then be accessed as if it were a simple array of bits; random or sequential access is possible using normal indexed addressing. More sophisticated functions, such as keyed files or data base systems, can be provided with user-level packages built using segments.

The input/output mechanism defined by the S-1 architecture utilizes small buffer memories shared with external I/O processors. Access to these buffers is allowed through io-segments, which are special kinds of segments created with a privileged operation. Like ordinary segments, they can be made directly addressible as part of the memory of a domain. (Although, the architecture requires that they be accessed with special instructions.) In this way, access to physical I/O devices can be controlled by granting access to the io-segment for the I/O processor that handles the device.

Occasionally, an I/O processor will control a number of different devices that are to be accessed by separate tasks. For example, the I/O processor may control a terminal multiplexer, or a disk or magtape controller. In such a case, a server task can be constructed that demultiplexes the I/O and communicates with the user tasks by means of message channels. Because the kernel does not internally buffer messages, I/O done in this fashion can be made very efficient. These control programs are not provided as an integral part of the kernel. Indeed, except for control of devices which contain storage system volumes, all such support is to be performed by user-level utilities.

It is useful to note, however, that standard protocols will be developed outside the kernel for servers supporting general classes of devices (terminals, graphics displays, disks and magtapes). Device simulation such as replacing a file with a terminal could be performed simply by replacing the server task.

Dynamic Reconfiguration

The kernel allows physical processors and memory to be added or removed from the system configuration at runtime. It is not necessary to "crash" the system to change the attachment of physical resources.

CURRENT STATUS

Several facilities described in this document have not yet been incorporated into the existing Amber kernel which currently runs on the S-1 Mark IIa Uniprocessor. Each kernel gate which does not yet exist is so labeled in the description of that gate later in this document.

The following facilities are not in the current Amber kernel:

1. message channels. This includes the `create_message_channel_`, `connect_`, `disconnect_`, `identify_caller_`, `send_`, `receive_`, `receive_packet_info_`, `flush_channel_`, and `monitor_task_` gates.
2. Extended objects. This includes the `seal_object_` and `unseal_object_` gates.
3. Quota. This includes the `set_default_volume_`, `set_default_volume_domains_`, `create_quota_account_`, `delete_quota_account_`, `set_quota_limit_`, and `list_quota_` gates.
4. Dynamic reconfiguration. This includes the `add_processor_`, `delete_processor_`, `add_memory_`, and `delete_memory_` gates.
5. Special segments. This includes the `create_special_segment_` gate.
6. The `invoke_domain_` gate.
7. The `move_object_` gate.

The lack of special segments and message channels has necessitated the implementation of a temporary terminal input/output facility. The gates that are part of this temporary mechanism are so marked in the documentation.

SUMMARY OF OBJECT CLASSES AND ACCESS MODES

Below is a complete list of the different classes of objects, and the modes of access permitted for each. The individual modes are ordered corresponding to their position in the bit mask representing the a mode set. The modes are given below according to the defined ordering. (For example, the first mode corresponds to bit zero of the mask; the second, the next to bit one; and so forth.)

All objects share certain properties and access modes. The common modes are given here. They are the first modes of all objects; modes unique to the various classes are listed below under the individual headings for the class.

get	allows the contents of the property list of an object and the attributes of the object held in the domain to be examined.
put	allows entries on the property list of an object to be changed.
listen	allows broadcast events associated with the object to be declared.
broadcast	allows events associated with the object to be signalled.
reserved-global-modes (4)	are four modes reserved for future enhancements to the system. In general, it is advisable for users not to grant these modes of access.

Domains

A domain object is a catalog. It contains entries for objects and links to objects, the names of the entries, and the attributes of the objects contained within it.

find	permits the domain to be searched for an entry with a particular name. This mode does not, itself, allow any of the attributes of the object to be obtained, or permit a listing of all names defined within the domain.
list	allows access to the "contents" of a domain. The names and status information about entries can be examined. Quota usage information about segments and subdomains can be obtained.
modify	permits the "contents" of the domain to be changed. Catalogued entries can be deleted, and their names changed. The access-control list of objects in the domain can be edited. Quota limits can be modified.
use	allows another domain to use access granted to the domain. Domains holding "use" access to another domain are, in effect, the agents or attorneys of the domain.
invoke	allows a copy of the domain to be created for the purpose of executing a task object that resides in the domain.

Segments

Segments are information containers, serving much the same function as files on other systems.

read	allows the contents of the segment to be read.
write	allows the contents of the segment to be modified. This includes truncation of the segment to control use of secondary storage records and control of the maximum length of the segment.
execute	allows the segment to be executed as a program. Normally, an executable segment should contain position independent code and be protected against write access.

Tasks

A task represents an executing program. The access modes of task objects allow them to be controlled and their state examined.

status	allows the state, status information, and scheduling parameters of the task to be examined.
writestate	allows the state of the task to be modified.
control	allows the task to be activated, started, stopped or killed.

Message Channels

A message channel is a bidirectional communication path. There are two capabilities for a single message channel, designating the opposing “ends”. These are called the “user” and “server” mode objects, but are functionally symmetric in all respects.

transmit	allows a task to grab the channel, and then send (receive) information to (from) the alternate end.
----------	---

Dummy Objects

A dummy object is merely a catalog entry for which no special operations are defined. The only state associated with a dummy object is its system maintained status attributes and property list. It is intended that dummy objects be used as building blocks for extended type objects requiring no special representation or place-holders for other non-protected user-defined objects.

Extended Type Objects

Objects tagged with a seal at the time of creation are controlled by their type manager (the sealing domain). The system places an interpretation on the common three modes of an extended type object, that is, get, put, listen, broadcast and the reserved modes. The remaining modes are not defined by the system and may be put to any use by the type manager of the object. (Type managers should, however, observe the rule that modes be positive, that is, the presence of a mode allows the user to perform more — not fewer — operations on the object than if the mode were not present.)

METHOD OF INTERFACE DEFINITION

A design goal of Amber is to allow the user a choice of programming language with which to write system programs. For this reason, the interfaces are defined using an abstract formulation which does not correspond to any particular programming language. Instead, it uses a limited set of constructs selected from major programming languages now in existence.

It will be possible to define a simple translation from the interface definitions to declarations of suitably rich languages. Whenever a language implementation uses the standard runtime environment and data representations assumed, programs written in that language can directly call the kernel routines in the manner indicated by this specification. In particular, the implementations of Pascal and Ada will be compatible with the kernel. Languages using a nonstandard environment or nonstandard data representations (e.g. LISP) will require wrappers for the kernel procedures.

Interface Definitions

An interface definition consists of a textual description of its function, a sample call labeling the parameters, a list of the parameters, and a list of exceptional conditions.

The description of the interface indicates the operation which is performed and the relationship to other interfaces. Specific details of the kernel function, beyond that given in the preceeding overview, are included. This section also contains any usage notes which may be appropriate.

The sample call gives a stylized procedure or function invocation. This gives the name of the interface, the name of each formal parameter, the order of the parameters, and, in the case of a function, the name of the value assigned the value returned by the function. Consider the following examples.

```
time = clock_read_ ()  
wait_ (event_id, time_out, status)
```

The first is a function invocation. The interface has no parameters and returns a value, called time. The second is a procedure invocation, with three parameters.

The parameter section enumerates all of the parameters and describes their usage. A header line gives the name of the parameter, its data type and its parameter type. A textual description follows.

The parameter type denotes the parameter's mode (value or reference) and access (input, output, or input-output). These keywords are given for all explicit parameters. If the access is omitted, then value is assumed. The keyword, result, denotes a function return value. (Note that kernel functions are restricted to returning only simple scalar values such as integers, status codes or boolean flags.)

The description of a parameter includes all information necessary to understand the use of the parameter. In particular, the description of a capability parameter will give the modes of access required for the operation.

When a parameter uses a structure type that is unique to that routine, it is convenient to follow the parameter definition immediately with the definition of the structure type. This includes both the structure declaration and the list of field definitions, and is inserted into the list of parameters.

The exceptional conditions section enumerates unusual circumstances that may abort the operation, occur as a side effect or result in partial completion. With each interface, only those conditions which are peculiar to the operation, which may occur during normal execution, or have a special significance to the operation are given. Other common error conditions, such as access violations, are listed at the end of this document in the "Common Exceptions" section.

Several techniques are used to describe the conditions. First of all, a logical predicate involving the return values may be given. The condition is assumed to exist when the predicate evaluates to true. Second, the name of a hardware or software trap may appear. This indicates that the trap has occurred. Third, if the name of a condition appears, it indicates that the condition is signalled. Fourth, the name of an status code may be given. In this case, the standard condition error has been signalled with a parameter equal to this code. In the last two

cases, additional parameters to the signal operation may serve to describe the cause of the exception.

Type Definitions

Two methods are used to define the types of routine parameters or the fields of data structures. The type may be specified as one of several primitive classes, or a type name may be used which refers to a primitive class definition.

All scalar types, i.e. integer subranges or enumeration types, are defined essentially as if they were primitive types. That is, their names are introduced, and the properties of the type given in a textual description. No specific representation is given. For each standard language, a set of declarations or macros for the types must be constructed. The standard scalar types are given in the "Standard Data Types" section below.

A character string is defined using a type definition which gives the length of the string. There are two formats.

```
string    string ( length )
```

The first format is used only for the type of a parameter and indicates that the string is of arbitrary length. The length of the actual parameter is passed along with its value. The second format can be used both for parameters and fields. The length value may be either a nonnegative integer constant or in the case of a field, the name of a preceding field which contains the actual length of the string.

Pointer types have two basic formats, each which may be qualified by the optional "relative" keyword:

```
pointer ( (type name) )    relative
pointer
```

In the first format, the type of the data pointed to is specified. This asserts that the program is incorrect if the pointer does not address an item of that type. In the second format, the target type is not specified, and the pointer may point to data of any type. Typically, the latter format is used when the value is a simple address in memory. The "relative" prefix indicates that the pointer value does not contain an absolute address, but rather a self-relative displacement for the addressed item.

Array types are defined as follows:

```
array [ lower-bound .. upper-bound ] of element-type
```

The bounds may be given as integer constants or, in the case of an array which is a field of a record, the name of a preceding field which contains the bound of the array. If a bound is not fixed, an asterisk ("*") is used. In this case, the array parameter is followed by an implicit parameter which is the actual bound.

Data structures are defined using the format of the Amber dialect of Pascal. A record type definition is given defining the name of the record type and each of the fields. After the declaration of the structure, textual descriptions of the usage of each field is given.

An array of structures may be defined by following the structure type name with a parenthesized bound list as in an array definition.

Type names are introduced in one of several ways. First, the type may be explicitly defined in a separate section, much like a routine description. This is used primarily for widely used data structures. Second, the definition may be implicitly given as the type of a parameter or field. The definition is interspersed with the parameter or field definition. This is done for types which have only a single reference. Third, the type name may be the name of one of the primitive types whose properties are described below.

Status Code Definition

The codes which are returned by kernel routines or contained in messages signalled with the error condition serve to indicate the success, failure or status of the operation performed. As the use of status codes is a common technique used by both system and applications programs, a general purpose mechanism has been defined for constructing error tables.

The value of a code is in fact an encoded reference to a table which contains information about the cause of the error. At present, only textual messages describing the situation will be available; however, more advanced information could be added at a later time without difficulty. Only the absolute value of the code is meaningful to identify the error. Values less than zero indicate that an unrecoverable error has occurred, while values greater than zero denote warning or status information. Zero is defined to mean no error.

The encoding technique allows for references to a large number of different tables. Thus, while there is a single kernel table, applications may define an essentially unlimited number of special purpose tables. The significant point is that the codes from different sources may be intermixed, and the encoding allows the identify of the table to be reconstructed. Thus, one application may pass on codes that it obtains from lower level modules.

Utility routines are defined to create the tables, assign the values of the codes, and to extract information associated with a code from the table to which it refers. These routines run in the user domain, and are not a necessary part of the kernel support. Only the codes need to be known.

By convention, all codes are given symbolic values identifying the table in which they appear. For example, all kernel codes begin with the prefix "syscode.". Codes from application error tables may be assigned other names.

Additional details about this facility will appear in documentation of user support routines.

STANDARD DATA TYPES

The following are definitions of the standard data types considered primitive by the kernel.

boolean	is a logical flag. The constants true and false are defined.
integer	designates a single-word integer value. The exact range of this type is not specified, and therefore the program should not rely on the implementation dependent limits.
unsigned_integer	describes a non-negative, integer, counting value. The range is left imprecisely defined as for the integer type.
segment_size	gives the type of value denoting the size of a segment. The range allows for the maximum number of addressable units within a segment. It should be used for values giving segment or data structure sizes.
segment_offset	gives the type of an address offset. The range allows for all offsets in units of addressable units.
signed_segment_offset	gives the type of a displacement within a segment.
bit_offset	gives the type of an absolute bit offset within a segment.
storage_unit	is a data type which packs like an addressable unit. An array of some number of storage units will have the size of that number of contiguous units. Two constants are associated with this type. <i>bits_per_unit</i> gives the number of bits in a storage unit; <i>max_segment_size</i> gives the maximum number of storage units in a segment.
char_index	is the type of an integer which can be used as the index of any string. It allows for an offset of -1 and of the value which is one greater than the maximum string length.
system_time	is a date/time stamp. They can be used to represent intervals, displacements or absolute values. The convention for representing absolute values uses a time origin of 0:00, January 1, 1900. The times are given in units of <i>system_time_second</i> ticks per second. The constant, <i>before_time</i> , denotes the "negative infinity" time; the constant, <i>after_time</i> is "positive infinity"
version_id	gives the type of a structure version number. Most structures in Amber which may last from task to task or system to system are given version numbers. This allows the software to recognize structures which may have been created by an old version.
unique_id	denotes a unique value. They are used, for example, to identify objects in the storage system, but can be used for similar purposes by the user. A kernel procedure is provided that returns different values on each call. Thus, any two unique id's which are the same, must denote the same object. The representation of an unique identifier is an unsigned integer double-word. The constant <i>null_unique_id</i> (zero) is used to denote an undefined value of this type.
object_name	is the type of the name of an object catalogued in a domain. Such names are character strings containing from 1 to 48 of the 95 printable ASCII characters. Trailing blanks are not significant. The constant <i>max_object_name_length</i> is defined and gives the current name length limit.
hierarchy_depth	is a constant which gives the maximum number of levels of names of an object cataloged in the storage system. The current value is 12, which allows for 11 levels of domains and the name of the terminal object itself.
entry_id	A <i>entry_id</i> value gives the unique identifier of a object or link cataloged in a domain.

When an entry id alone is used in a kernel operation, it denotes the object (or target object in the case of a link) to be operated on. When both a domain and entry id are used in a kernel operation (such as `set_names_`) it is the entry itself which is to be operated on. A special value, `null_id`, is defined; it refers to no entry and can be used as a place holder or undefined value.

access_mode_set

is the type of the access mode set of an object. The representation is a bit string of maximum length 32. The meaning of each individual bit is dependent on the particular type of object; however, the requirement is made that a bit turned on represent an increase in privilege. The constants, `full_mode_set` and `null_mode_set`, are defined and give the set of all modes and the empty set respectively. (In addition, constants will be defined for the modes of each primitive class of objects.)

object_class

is the type of the set of all primitive classes of objects. The members of this set are defined by the constants: `domain_object`, `segment_object`, `channel_object`, `task_object`, `sealed_object` and `dummy_object`.

status_code

is the type for a standard system status code. Such codes are returned by procedures to indicate the success or failure of a requested operation. The representation is a large integer encoding a reference to a table which contains a description of the error or other condition. The absolute value only is meaningful to identify the message. Values less than zero are used to denote unrecoverable errors; values greater than zero denote warning or status information. Zero is defined to mean no error.

event_id

is the type of event identifiers used in the broadcast mechanism. The internal representation is an unsigned integer double-word. The constant, `null_event_id` (zero), denotes an undefined value of this type.

wakeup_id

is the type of a wakeup index. It is represented as an unsigned integer single-word. The maximum wakeup index is defined on a per installation basis. The constant, `null_wakeup_id` (zero), denotes an undefined value of this type.

processor_id

is a subrange type which gives the numeric identifiers of the different processors. This value lies in the range 0 to `max_processor_number`.

processor_set

is a type denoting a set of processors. If element `n` of the set is on, then processor `n` is selected. The constant `all_processors` gives the set of all processors.

task_priority

is a subrange type giving scheduling priority values. A priority value lies in the range 0 to `max_priority`.

GATE: `create_domain_`

This creates a domain object. The owner of the new domain is set to the owner of the current domain. The owner may be changed by a subsequent, privileged operation.

```
id = create_domain_ (domain, name, seal, volume, modes)
```

where:

domain: `entry_id` input

designates the domain in which the subdomain is to be created. (If null, then the current domain is used.) "Modify" access to this domain is required.

name: string input

if non-blank, this gives the name to be given to the new domain. This must not be a duplicate of any other name in the domain. If the name is blank, then the new domain has no name and can only be referenced by id.

seal: `entry_id` input

if non-null, designates the domain as a protected, extended type object whose representation is a domain. Access to the unseal the representation is allowed only to this domain (or its agents). No special access to the seal domain is required.

volume: `unique_id` input

designates the volume on which the domain is to reside. If null, then the default domain volume is taken from the parent (containing) domain. If the containing domain resides on a second class volume, this must be the same volume.

modes: `access_modes` input

if non-null, this gives the modes of access to be granted to the containing domain. An access-control list entry for the domain is created to permit the access; this entry may be modified or deleted to change the access, subsequently.

id: `entry_id` result

is assigned the id of the domain created.

Exceptional conditions:

syscode.max_hierarchy_depth_reached

An attempt has been made to create a domain at the deepest level of the hierarchy. No objects may be appended without exceeded the maximum permitted depth, so the operation fails.

GATE: set_domain_owner_

This is used to set the principal which is responsible for a domain. Since a domain is considered to have implicit rights to use access granted to its owner, this operation greatly affects the privileges of a domain and is therefore privileged.

set_domain_owner_ (domain, owner)

where:

domain: entry_id input

designates the domain for which the owner is to be set. No special access to this domain is required.

owner: entry_id input

designates the domain which is to be identified as the owner of the other.

GATE: move_object_

This moves an object from one domain to another. In addition, the volume on which segment and domain objects reside may be changed.

The object, its access control list and its attributes are transferred to the new domain. The transplanted object retains the names of the original. (If a name conflict would result, an error is reported and the move is not performed.) An address which is assigned to the object is deleted, so that a new mapping must be performed if required. The uid of the object is preserved; however, all outstanding links to the object are invalidated.

Movement of an object must maintain the following constraint: if a domain resides on a second-class volume, then all inferior objects must reside on the same second class volume; if a domain resides on a first-class volume, then inferiors may reside on any first- or second-class volume.

Note that when a domain is moved from one volume to another, the default volumes for the creation of inferior objects may become inconsistent with respect to the volume class distinction. The user should be careful to change the defaults if necessary.

This operation may be used to change only the volume on which an object resides, by specifying the same domain as the source and target of the move.

This gate has not yet been implemented.

move_object_ (old_domain, object, new_domain, new_volume, new_object)

where:

- old_domain: entry_id input**
gives the domain in which the object originally appears. "Modify" access to this domain is required.
- object: entry_id input**
is the id (within old_domain) of the object to be moved. The names and access control list are copied from this link. No special access to the object, itself, is required.
- new_domain: entry_id input**
designates the domain to which the object is to be moved. If this is the same as the old domain — i.e., only the volume of the object is being changed, then no further access is required; if this is a different domain, then "modify" access is necessary.
- new_volume: unique_id input**
designates the volume to which the object is to be moved. (If null, then the default volume from the new domain is used.) No special access is required for the volume; however, there must be sufficient quota within the containing domain to hold the object on the volume. (This is ignored for objects other than a segment or domain.)

Exceptional conditions:

syscode.name_duplication

The name of the object being moved conflicts with the name of an existing entry in the destination domain. This error does not occur when the domain containing the object is not being changed.

syscode.hierarchy_depth_exceeded

The number of domain levels which would have been created by the operation exceeded the maximum depth. For example, with a maximum depth of 12, a domain with 8 sublevels could not be moved into a domain at level 4.

GATE: `lookup_`

This searches a domain for an entry (object or link) with a specified name, and returns the link by which it can be referenced in other operations.

`id = lookup_ (domain, name)`

where:

`domain: entry_id` input

designates the domain to be searched. (If null, then the current domain is used.) "Find" access to this domain is required.

`name: string` input

gives the name of the entry to be searched for. It must not be the null string.

`id: entry_id` result

is assigned the identifier of the entry found.

Exceptional conditions:

`id = null_id`

indicates that no object was found with the specified name. This condition also occurs if one of the error conditions is signalled.

GATE: `list_entries_`

This is an enquiry operation that provides a list of all entries (objects or links) in a domain.

`list_entries_ (domain, entry_id_list_ptr, max_entries)`

where:

`domain`: `entry_id` input

designates the domain whose contents are to be listed. "List" access to the domain is required.

`entry_id_list_ptr`: pointer (`entry_id_list`) input

gives the address of an area into which the list of objects can be placed. The format of this structure is given below.

```

type entry_id_list = record
  version: version_id;
  n_ids: unsigned_integer;
  entry: array [1..n_ids] of record
    id: entry_id;
    primary_name: object_name;
    link: boolean;
  end;
end;
```

where:

`entry_id_list.version`

gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

`entry_id_list.n_ids`

gives the count of the number of objects returned.

`entry_id_list.entry`

is an array that contains the entries. The array is not ordered in any meaningful way.

`entry_id_list.entry.id`

gives the id of the object or link.

`entry_id_list.entry.name`

gives the text of the primary name of the object or link. If the object has no names, this field will be blank.

`max_entries`: integer input

gives the maximum number of name list entries that have been allocated.

Exceptional conditions:

`entry_id_list.n_ids > max_entries`

occurs when the structure area is too small to hold all of the entries in the directory. All available entries are filled in.

GATE: `list_entries_status_`

This is an enquiry operation that provides a list of all entries (objects or links) in a domain and returns status information for each entry.

`list_entries_status_ (domain, entry_status_list_ptr, max_entries, path_list, n_paths)`

where:

- `domain`: `entry_id` input
designates the domain whose contents are to be listed. "List" access to the domain is required.
- `entry_status_list_ptr`: pointer (`entry_status_list`) input
gives the address of an area into which the list of objects can be placed. The format of this structure is given below.
- `max_entries`: integer input
gives the maximum number of name list entries that have been allocated.
- `path_list`: array [1..*] of `entry_id` input
gives a list of domains to be used to obtain the "get" access required to return status information for the entries. If no paths in the list provide "get" access to an entry, status information will not be returned for that entry.
- `n_paths`: unsigned integer input
gives the number of valid entries in the `path_list` array.

```

type entry_status_list = record
  version: version_id;
  n_ids: cardinal;
  entry: array [1..n_ids] of record
    id: entry_id;
    primary_name: object_name;
    code: status_code;
    modes: access_mode_set;
    info: object_status_type;
  end;
end;
```

where:

- `entry_status_list.version`
gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.
- `entry_status_list.n_ids`
gives the count of the number of objects returned.
- `entry_status_list.entry`
is an array that contains the entries. The array is not ordered in any meaningful way.
- `entry_status_list.entry.id`
gives the id of the object or link.
- `entry_status_list.entry.primary_name`
gives the text of the primary name of the object or link. If the object has no names, this field will be blank.
- `entry_status_list.entry.code`
If this is non-zero, then an error was encountered while trying to gather status information for this entry. Some, or all of that information may be incorrect depending on

the error.

entry_status_list.entry.modes

gives the access obtained to this object using the first domain in **path_list** that provided at least “get” access. If “get” access is not in this set, the status information is not valid for this entry.

entry_status_list.entry.info

This is a record containing status information for the object. See the description of the **object_status_gate** for details.

Exceptional conditions:

entry_status_list.n_ids > max_entries

occurs when the structure area is too small to hold all the entries in the directory. All available entries are filled in.

GATE: list_entries_by_name_

This is an enquiry operation that provides a list of all names of objects or links in a domain. The caller may request a list of all the names or only those whose names begin with a certain string. This feature assists user level operations which provide wild card matching in names or name completion.

Objects or links in the domain that have no non-blank names will not appear in the list returned by this entry.

```
list_entries_by_name_ (domain, prefix, case_sensitive, link_name_list_ptr,
                      max_entry_names)
```

where:

domain: entry_id input

designates the domain whose contents are to be listed. "List" access to the domain is required.

prefix: string input

gives the name prefix to be searched for. For example, a prefix of "a" selects all links whose names begin with the letter "a". If a zero-length string is used, all entries including those with no explicit names, are deleted.

case_sensitive: boolean input

indicates that the case of alphabetic characters should be considered when matching the prefix. If this is set to false, a prefix of "a" matches "a" or "A"; if true, "a" matches only "a".

entry_name_list_ptr: pointer (entry_name_list) input

gives the address of an area into which the list of names with the specified prefix can be placed. The format of this structure is given below.

```
type entry_name_list = record
  version: version_id;
  n_names: unsigned_integer;
  entry: array [1..n_names] of record
    name: object_name;
    id: entry_id;
  end;
end;
```

where:

entry_name_list.version

gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

entry_name_list.n_names

gives the count of the number of names returned.

entry_name_list.entry

is an array that contains the names/entries selected. The array is ordered in alphabetic order according to the names.

entry_name_list.entry.name

gives the text of the name of the object or link.

entry_name_list.entry.id

gives the id of the object or link corresponding to the selected name. Since an entry may have several names, it may be listed several times under different names.

max_entry_names: integer input

gives the maximum number of name list entries that have been allocated.

Exceptional conditions:

entry_name_list.n_names > max_entry_names

occurs when the structure area is too small to hold all names selected by the search. All available entries are filled in.

GATE: `link_info_`

This is an enquiry function that returns status information kept for a link. If applied to an object entry, default information is returned.

```
link_info_ (domain, id, link_info, max_paths)
```

where:

`domain`: `entry_id` input

designates the domain containing the link. (If null, then the current domain is used.) "List" access to this domain is required.

`id`: `entry_id` input

designates the entry for which the status is to be returned. No special access to the target of the object is required.

`link_info`: `link_info_type` reference input-output

is a structure whose fields are assigned the status information. A version field must be set on input and specifies the version of the structure to be used.

`max_paths`: `unsigned_integer` input

gives the maximum number of access path entries for which space has been allocated in the `link_info` structure. Setting this parameter to zero causes the access path not to be returned.

```
type link_info_type = record
  version: version_id;
  time_created: system_time;
  time_entry_modified: system_time;
  time_entry_dumped: system_time;
  modes: access_mode_set;
  target: entry_id;
  n_paths: unsigned_integer;
  access_path: array [n_paths] of entry_id;
end;
```

where:

`link_info.version`

gives the version of the structure. Currently, only one version is supported and this value should be set to 1 on input.

`link_info.time_created`

gives the time at which the link or object was first created.

`link_info.time_entry_modified`

gives the time at which the entry for the link or object was last modified. This includes creation or any alterations to its names.

`link_info.time_entry_dumped`

gives the time at which the entry information for the link or object was last dumped by the backup system. A value of *before_time* is returned if the information has never been dumped.

`link_info.modes`

gives the limit on the modes of access which may be obtained with this link. If the id denotes a normal object, rather than a link, this value is set to the universal mode set.

`link_info.target`

gives the id of the object that the link points to. If the id denotes a normal object, rather than a link, this value is set to the null id. This is created for the sole purpose

of identifying the target. The caller should delete the link when done.

link_info.n_paths

gives the number of valid entries in the access_path array, below.

link_info.access_path

gives the access path used to obtain access to the object. This is an array whose first element gives the principal of the domain (owner) and whose last element is compared against the access control list of the object. Intermediate entries are those which establish a chain of "use" access to the last entry. In the case of an object entry, the containing domain is returned. (The ids are for links created for the sole purpose of identifying the principal; it has no name and allows no access. The caller should delete the link when done.)

Exceptional conditions:

link_info.n_paths > max_paths

occurs when the number of access path entries exceeds the space allocated; n_paths gives the correct number of entries, and the rest of the information structure is filled in.

GATE: get_pathname_

This determines the location of an object in the storage system. Its **pathname** is returned; for each component of the path, both the uid and the primary name (if available) of the entry is supplied.

get_pathname_ (object, pathname)

where:

object: entry_id input

designates the object whose location is to be determined. No special access modes are required for this operation.

pathname: object_pathname reference output

is set to contain the pathname information of the object.

```

type object_pathname = record
  version: version_id;
  depth: unsigned_integer;
  known_depth: unsigned_integer;
  status: status_code;
  component: array [1..Max_Hierarchy_Depth] of record
    name: object_name;
    id: entry_id;
  end;
end;
```

where:

object_pathname.version

gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1.

object_pathname.depth

gives the depth of the object in the domain hierarchy. For example, the depth of the root is 0, and the depth of an object in the root domain is 1. This value can never be greater than the constant, **Hierarchy_Depth**. Only this number of entries in the component array is used.

object_pathname.known_depth

gives the depth to which the names of the object are known. In the case of a link to an object, it may not be possible to determine the full pathname of the target for two reasons. First, the subtree of the hierarchy containing the object may be on an unmounted volume; only the names of the components in domains which are not offline can be supplied. Second, the object or a subtree containing the object may have been deleted since the link was created; only the remaining components can be identified. The status field indicates the reason.

object_pathname.status

indicates any problem in locating an object. If 0, then the full pathname is successfully returned. If **syscode.object_offline**, then part of the hierarchy containing the object is not mounted. If **syscode.entry_not_found**, then the object has been deleted.

object_pathname.component

is an array contain the name and unique id of each component of the pathname of the object. Entry 1 corresponds to the domain which appears in the root, and entry **depth** corresponds to the deepest leaf that is the object itself.

object_pathname.component.name

gives the primary name of a component. This is all blank if the entry has no name, or

if its depth is greater than *known_depth*.

`object_pathname.component.id`

gives the unique identifier of the object. This can always be returned.

GATE: `list_names_`

This operation returns all of the names on a entry.

`list_names_ (domain, id, name_list_ptr, max_names)`

where:

`domain`: entry_id input

designates the domain in which the entry appears. (If null, then the current domain is assumed.)
"List" access to this domain is required.

`id`: entry_id input

designates the entry whose names are to be listed. No special access is required.

`name_list_ptr`: pointer (name_list) input

is the address of the area into which a structure containing the list of names is placed. The format of this structure is given below.

```

type name_list = record
  version: version_id;
  n_names: unsigned_integer;
  names: array [1..n_names] of object_name;
end;
```

where:

`name_list.version`

gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

`name_list.n_names`

gives the number of names on the entry.

`name_list.names`

is an array that contains the names on the entry. The first name listed is the primary name on the entry.

`max_names`: integer input

gives the maximum number of names that have been allocated.

Exceptional conditions:

`name_list.n_names > max_names`

occurs when the entry has more names than space was allocated for. As many names as possible are filled in.

GATE: `set_names_`

This sets the list of names for an entry. The order of the names is significant. The first name is the primary name, which is returned by several other operations, and the order of the other names is preserved by the `list_names_` operation.

```
set_names_ (domain, id, names, n_names, validation_time)
```

where:

domain: `entry_id` input

designates the domain in which the entry appears. (If null, then the current domain is assumed.)
"Modify" access to this domain is required.

id: `entry_id` input

designates the entry whose names are to be changed. No access is required to the target object.

names: array [1..*] of `object_name` input

is the array of names to be set. The name order follows increasing subscripts. Only "n_names" elements are used.

n_names: unsigned integer input

gives the number of names in the "names" array. If zero, then the entry is left without a textual name and can only be referenced by id.

validation_time: `system_time` input

is an absolute time used to validate the change to the name list. If the list being set is derived from the existing list (e.g. from `list_names_`), the validation time is the time at which the original list was obtained. If the domain information for the entry (possibly including the names) was modified after this time, then the operation is not attempted. A value of "After_Time" suppresses this check.

Exceptional conditions:

`syscode.name_list_empty`

is an error which occurs if the number of names is zero or less.

`syscode.validation_fails`

is an error signal which indicates that the validation check failed.

GATE: `delete_entry_`

This operation deletes an object or link from a domain.

If there is a memory mapping associated with the entry (e.g. a segment or a link to a segment), then the memory slot is freed and may be reused.

No restriction is placed on the deletion of protected objects. Only the access to the containing domain is considered.

If the object being deleted is a domain, then all inferior objects are deleted. This operation succeeds regardless of the access available to inferior domains.

`delete_entry_ (domain, id)`

where:

domain: `entry_id` input

designates the domain in which the entry appears. (If null, then the current domain is assumed.)
"Modify" access to this domain is required.

id: `entry_id` input

designates the entry that is to be deleted.

GATE: `create_link_`

This operation creates a link to an object.

`id = create_link_ (source_domain, source_id, modes, path, target_domain, target_name)`

where:

source_domain: `entry_id` input

designates the domain containing the object or a link to the object. (If null, then the current domain is used.) "Find" access to this domain is required.

source_id: `entry_id` input

gives the identifier of a entry within the above domain. If this entry is an object, then the link is made to point to the object; if the entry is a link, then the new link is made to point at the target object.

modes: `access_mode_set` input

specifies a limit on the modes of access to be used with this link. Any set of modes may be specified, but their exercise is controlled by the access control list of the object. These modes may not be changed.

path: `entry_id` input

is a link to the principal domain which is the final component of the of the access path. No access is required to specify this domain. If path is null, then the target object (as a single component) must be a legal path for the target domain, i.e. it must be either the target domain or its owner. If path is a link which was obtained from a revocable seal, then an error occurs. Otherwise, an access path is created using this as the final component and the access path to it as the preceding components. For example, to designate an access path O.G, where O is the owner of the current domain, and G is some principal that O has access to. One must link to O specifying a null path and then link to G specifying the O link as a path; the G link may then be used as an access path for O.G.

target_domain: `entry_id` input

designates the domain in which the new link is to be placed. (If null, then the current domain is used.) "Modify" access to this domain is required.

target_name: `string` input

if non-blank, this gives the name to be given to the new link. This must not be a duplicate of the name of any other name in the domain. If the name is blank, then the link has no name and must be referenced by uid alone.

id: `entry_id` result

is assigned the identifier for the new link.

Exceptional conditions:

`syscode.invalid_access_path`

is an error signal which occurs (1) if the path is null and the target is not a valid first component of an access path, (2) if the link was derived from a revocable seal, or (3) if the path refers directly to an object rather than a link.

GATE: create_link_from_acl_

This operation creates a link to an object. Unlike `create_link_`, where a specific access-path is specified, the caller supplies a list of possible access-paths with which to attempt to obtain access. This operation selects once such entry which provides the requested access to the object.

```
id = create_link_from_acl_ (source_domain, source_id, modes, paths, n_paths, target_domain,
                           target_name)
```

where:

source_domain: entry_id input

designates the domain containing the object or a link to the object. (If null, then the current domain is used.) "Find" access to this domain is required.

source_id: entry_id input

gives the identifier of a entry within the above domain. If this entry is an object, then the link is made to point to the object; if the entry is a link, then the new link is made to point at the target object.

modes: access_mode_set input

specifies a limit on the modes of access to be used with this link. Any set of modes may be specified, but their exercise is controlled by the access control list of the object. These modes may not be changed.

paths: array [1..*] of entry_id input

is an array of entry ids for principal domains giving possible access paths. (Only the first `n_paths` entries are used.) The first (lowest subscript) for which there is an access control list entry supplying the required modes is selected. This is used to form the access path in the same manner as is used in the `create_link_` operation; the same restrictions applies.

n_paths: unsigned_integer input

gives the number of entries in the above array that are used.

target_domain: entry_id input

designates the domain in which the new link is to be placed. (If null, then the current domain is used.) "Modify" access to this domain is required.

target_name: string input

if non-blank, this gives the name to be given to the new link. This must not be a duplicate of the name of any other link in the domain. If the name is blank, then the link has no name and must be referenced by uid alone.

id: entry_id result

is assigned the identifier for the new link.

Exceptional conditions:

rcode.invalid_access_path

is an error signal which occurs if the path cannot be legally used because the path was derived from a revocable seal, or if the path refers directly to an object in the current domain rather than a link.

GATE: `list_acl`

This returns the contents of the access control list for an object.

```
list_acl (domain, objlink, acl_ptr, max_entries)
```

where:

`domain`: `entry_id` input

designates the domain in which the object resides. (If null, then the current domain is assumed.)
The operation is allowed if "list" access to the domain is available.

`objlink`: `entry_id` input

is the id, within the above domain, of the object whose ACL is to be listed. An error occurs if this id refers to a link.

`acl_ptr`: pointer (acl) input

is the address of an area into which the access control list is to be placed. The format is given below.

```
type acl = record
  version: version_id;
  n_entries: unsigned_integer;
  entry: array [1..n_caps] of record
    modes: access_mode_set;
    principal: entry_id;
  end;
```

where:

`acl.version`

gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

`acl.n_entries`

gives the number of entries in the access control list.

`acl.modes`

gives the modes of access held by the associated principal.

`acl.principal`

is the id of a link to the principal which holds access to the object. (The link is created for the sole purpose of identifying the principal; it has no name and allows no access. The caller should delete the link when done.) If this is the null id, then the entry is the default for all users.

`max_entries`: integer input

gives the maximum number of access control list entries that may be returned.

Exceptional conditions:

`syscode.not_an_object`

is an error signal which occurs if the id specified refers to a link rather than an object.

`acl.n_entries > max_entries`

occurs if there are more access control list entries than space has been allocated for. As many entries as possible are filled in.

GATE: `set_acl_`

This set the access control list of an object.

```
set_acl_ (domain, id, acl_ptr, validation_time)
```

where:

`domain`: `entry_id` input
designates the domain in which the object resides. (If null, then the current domain is assumed.)
"Modify" access to the domain is required.

`id`: `entry_id` input
is the id, within the above domain, of the object whose ACL is to be set.

`acl_ptr`: pointer (`acl`) input
is the address of the access control list for the object. The format is given below.

```
type acl = record
  version: version_id;
  n_entries: unsigned_integer;
  entry: array [1..n_caps] of record
    modes: access_mode_set;
    principal: entry_id;
  end;
```

where:

`acl.version`
gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

`acl.n_entries`
gives the number of entries in the access control list.

`acl.modes`
gives the modes of access to be granted to the associated principal.

`acl.principal`
is the entry for the principal domain which is to be granted access. If this value is `null_id`, then the entry is the default for all users.

`validation_time`: `system_time` input
is an absolute time used to validate the change to the name list. If the list being set is derived from the existing list (e.g. from `list_acl_`), the validation time is the time at which the original list was obtained. If the domain information for the link (possibly including the ACL) was modified after this time, then the operation is not attempted. A value of "After_Time" suppresses this check.

Exceptional conditions:

`syscode.duplicate_acl_entry`
occurs if the same principal is named in more than one access control list entry.

`syscode.not_an_object`
is an error signal which occurs if the id specified refers to a link rather than an object.

`acl.n_entries > max_entries`
occurs if there are more access control list entries than space has been allocated for. As many entries as possible are filled in.

GATE: seal_object_

This operation creates a sealed object containing a link having the caller's access to some object (which may be another extended type object); this permits the domain which is allowed to unseal the object to obtain a copy of the protected link. Since the link will have the originator's access, the operation may be used to pass access from one domain to another without altering the access control list of the object. The sealing operation may be specified to be revocable, in which case deletion of the sealed object results in revocation of access from any link derived from the sealed object.

The sealed object created by this operation is equivalent to an extended type object created with `create_segment_` (`create_domain_`, etc.), except that the representation is not strictly hidden. It may be given an access control list holding extended access.

The sealing mechanism, including this gate, has not yet been implemented.

```
id = seal_object_ (source_domain, source_id, seal, revoke, modes, path, target_domain,  
target_name)
```

where:

source_domain: entry_id input

designates the domain containing the object or a link to the object. (If null, then the current domain is used.) No special access to this domain is required.

source_id: entry_id input

gives the identifier of a entry within the above domain. If this entry is an object, then the link is made to point to the object; if the entry is a link, then the new link is made to point at the target object.

modes: access_mode_set input

specifies the modes of access which can be obtained with the unsealed link. Use of these modes is further limited by the access control list of the object. These modes may not be changed.

seal: entry_id input

designates the domain which is allowed to unseal the object. No special access to the this domain is required.

revoke: boolean input

if true, then the seal is revocable and deletion of the sealed object revokes access from links derived from it.

path: entry_id input

designates the principal domain which is the final component of the of the access path. No access is required to specify this domain. The full access path is derived as in the `create_link_` operation; however, it may not be null.

target_domain: entry_id input

designates the domain in which the new link is to be placed. (If null, then the current domain is used.) "Modify" access to this domain is required.

target_name: string input

if non-blank, this gives the name to be given to the new sealed object. This must not be a duplicate of any other name in the domain. If the name is blank, then the new object has no name and can only be referenced by id.

id: entry_id result

is assigned the identifier for the new sealed object.

Exceptional conditions:

`syscode.invalid_access_path`

is an error signal which occurs (1) if the path is null, (2) if the link was derived from a revocable seal, or (3) if the path refers directly to an object rather than a link.

GATE: `unseal_object_`

This unseals the representation of a protected, extended type object and makes it available for access. A link for the representation of the object is created and entered into a specified domain.

The sealing mechanism, including this gate, has not yet been implemented.

```
id = unseal_object_ (object, seal, domain, name, modes)
```

where:

object: `entry_id` input

designates the extended object to be unsealed.

seal: `entry_id` input

gives the domain which manages the object. (If null, then the current domain is assumed.) "Use" access to this domain is required.

domain: `entry_id` input

designates the domain into which the link for the object representation is to be placed. (If null, then the current domain is used.) "Modify" access to this domain is required.

name: `string` input

if non-blank, this gives the name to be given to the new link. This must not be a duplicate of any other name in the domain. If the name is blank, then the link has no name and can only be referenced by id.

modes: `access_mode_set` input

gives a limit on the modes of access to allowed on the representation through the link. If the sealed object is an extended type object (i.e. it was created with `create_segment_`, etc.), then these are the ultimate modes of access. If the sealed object holds a sealed link (i.e. it was created with the `seal_link_`), then these modes are intersected with the modes specified in the seal call.

id: `entry_id` result

is assigned the id of the newly created link.

Exceptional conditions:

syscode.type_mismatch

is an error signal. It indicates that the object could not be unsealed because the sealing domain did not match the actual sealing domain.

syscode.domain_no_use_access

is an error signal which indicates that access to "use" the sealing domain was lacking.

GATE: validate_access_

This verifies that modes of access that a domain has to an object are suitable to perform some operation. It is intended that this operation be used by extended type managers to verify that the caller has sufficient access to perform some operation.

```
validate_access_ (ref, class, seal, required_modes, actual_modes, code)
```

where:

- ref: entry_id input**
designates an entry for the object being referenced. The modes of access allowed through with this entry are computed and checked.
- class: object_class input**
gives the primitive type expected. This must match the class of the object (or the class of the representation of a typed object).
- seal: entry_id input**
gives the domain of the extended type manager of the object. If the object is not a typed object, then this must be a null value. If the object is an extended-type object, then this must reference the sealing domain specified at the time of creation of the object.
- required_modes: access_mode_set input**
gives the modes of access required. This must be a subset of the modes allowed to the current domain, or an error code is returned.
- actual_modes: access_mode_set output**
is assigned the computed modes of access.
- code: status_code output**
is assigned a code indicating whether or not the above tests succeed. The value is zero if all the tests succeed. Otherwise, the code value will be set to one of the values given below.

Exceptional conditions:

- code = syscode.entry_not_found**
indicates that the id does not correspond to that of any object catalogued in the current domain.
- code = syscode.type_mismatch**
indicates that the expected extended type of the object was not correct.
- code = syscode.not_a/an_(class)**
indicate that the primitive class expected did not match the actual class of the object. The code corresponding to the expected type is used, e.g. `syscode.not_a_segment` or `syscode.not_an_domain`.
- code = syscode.(class)_no_(mode)_access**
is signalled if the task does not have a particular mode of access to a non-typed object. The specific code for the class and mode is used, e.g. `syscode.domain_no_find_access` or `syscode.seg_no_read_access`.
- code = syscode.incorrect_access**
indicates that the task does not have a required access mode to a typed object. This code is used since there is no standard error message for the modes of an extended type.

GATE: `create_segment_`

This creates a segment object.

```
id = create_segment_ (domain, name, seal, volume, modes, max_length)
```

where:

domain: `entry_id` input

designates the domain in which the segment is to be created. The task must have “modify” access to the domain.

name: `string` input

if non-blank, this gives the name to be given to the new segment. This must not be a duplicate of any other name in the domain. If the name is blank, then the new segment has no name and can only be referenced by id.

seal: `entry_id` input

if non-null, designates the segment as a protected, extended type object whose representation is a segment. Access to unseal the representation is allowed only to this domain (or its agents). No special access to the sealing domain is required.

modes: `access_mode_set` input

if non-null, this gives the modes of access to be granted to the containing domain. An access-control list entry for the domain is created to permit the access; this entry may be modified or deleted to change the access, subsequently.

volume: `unique_id` input

designates the volume on which the object is to reside. If null, then the default segment volume is taken from the parent domain. If the containing domain resides on a second class volume, this must be the same volume.

max_length: `segment_size` input

gives the initial maximum length of the segment in quarter-words. The value must be less than or equal to 2^{*31} . (The maximum length is enforced the nearest integral page length.

id: `entry_id` result

is assigned the id of the newly created segment.

GATE: truncate_segment_

This operation truncates a segment object. The effect of truncation is to zero a specified amount of storage starting at a specified place in the segment. All full records within that area of the segment are discarded, thereby reducing the quota and physical storage utilization of the segment. However, if the segment is wired, the records are reinstantiated (containing zeros) as they may need to be referenced immediately. The name "truncate" refers to the common practice of zeroing the records at the end of the segment.

truncate_segment_ (segment, start, length)

where:

segment: entry_id input

designates the segment to be truncated. "Write" access to the segment is required.

start: segment_size input

designates the address within the segment where zeroing is to begin. This value is rounded down to the next record boundary (4096 quarter-words).

length: segment_size input

gives the length in quarter-words of the area to be zeroed.

GATE: `set_max_length_`

This operation sets the maximum length of a segment. Any attempt to access locations in the segment beyond the maximum length causes an addressing trap to occur, and hence provides a bounds-checking mechanism. Since the hardware only supports this checking to the nearest page boundry that is the granularity of the `max_length`. It is expessed in quarter-words anyway.

If the value specified for the maximum length is greater than the current size of the segment (maximum page number in use), then the segment is first truncated to the specified maximum length.

`set_max_length_ (segment, seg_max_length)`

where:

`segment`: `entry_id` input

designates the segment whose maximum length is to be set. "Write" access to the segment is required.

`seg_max_length`: `segment_size` input

gives the maximum length of the segment in quarter-words. The value must be less than or equal to $2^{*}31$. (The maximum length is enforced to the nearest integral page length.

Exceptional conditions:

`syscode.invalid_maximum_length`

is an error signal indicating that the maximum length value was out of the allowable range.

GATE: `create_special_segment_`

This creates a “special” segment. A special segment is defined by an implementation dependent descriptor which describes the object to be made accessible. For example, internal supervisor segments and io-segments may be accessed via special segments. Creation of these objects is a privileged operation.

The special segment mechanism, including this gate, has not yet been implemented.

`id = create_special_segment_ (domain, name, seal, descriptor, modes)`

where:

domain: `entry_id` input

designates the domain in which the object is to be created. The task must have “modify” access to the domain.

name: string input

if non-blank, this gives the name to be given to the new special segment. This must not be a duplicate of any other name in the domain. If the name is blank, then the special segment has no name and can only be referenced by id.

seal: `entry_id` input

if non-null, designates the special segment as a protected, extended type object whose representation is an special segment. Access to unseal the representation is allowed only to this domain (or its agents). No special access to the sealing domain.

descriptor: `special_seg_desc_type` input

gives the descriptor describing the internal object to be made accessible. The format of this descriptor is not defined at the moment.

modes: `access_mode_set` input

if non-null, this gives the modes of access to be granted to the containing domain. An access-control list entry for the domain is created to permit the access; this entry may be modified or deleted to change the access, subsequently.

id: `entry_id` result

is assigned the identifier of the link for the newly created object.

GATE: map_segment_

This operation sets the address of an entry. The effect is to map the (special) segment referenced into the into address space of the domain, and make it directly addressible by tasks executing in that domain. The modes of access allowed on locations within the mapped segment are exactly those allowed when the entry id is specified in a kernel call. (In effect, the address is just name for an object – the name used by the hardware.)

The entry may either be a segment object in the domain, or a link to a segment in the same or another domain. As there is a one-to-one correspondence between virtual memory slots and the entries (segments or links) in the domain, a segment may appear in the address space several times — at different addresses, and with different modes of access.

If there already exists a mapping for a link for a segment, this call may be used to change the address or size. The object is unmapped from the location it is in, then mapped into a new location. If the new location cannot be allocated, then the segment is left unmapped.

map_segment_ (domain, id, size, requested_base, address)

where:

domain: entry_id input

designates the domain which holds the link. (If null, then the current domain is used.) “Modify” access to the domain is required.

id: entry_id input

designates the entry for which a mapping is to be established. No particular modes of access are required, but whatever modes are available are allowed to instructions which reference locations corresponding to the segment.

size: segment_size input

gives the number of quarter-words of virtual memory to be occupied by the segment; this value is rounded to the next greater boundary required by the hardware and may be greater or less than the actual length of the segment. If a size of zero is specified, then the maximum length of the segment is used by default. For a special segment, this argument is ignored, and the system defined size is used.

requested_base: address_ input

if non-null, then this gives the requested base address at which to locate the object; if this is equal to *Null_Address*, then the kernel assigns an address on the basis of the size.

address: pointer output

is assigned the address at which the segment appears. If a specific address is requested by the caller, then this is set to that address. If the operation fails because a slot could not be found for the object, then a null pointer is returned.

Exceptional conditions:

syscode.duplicate_segno

is an error signalled if a requested address has already been allocated. (No such error occurs if the object is remapped over its original location.)

syscode.invalid_base_address

is an error signalled if the requested address is out of range, or is not consistent with the alignment requirements for the specified size.

GATE: `segment_address_`

This is an enquiry function that returns the address by which a domain can reference a segment. A mapping may be established for each link for a segment held by a domain. Thus, the address associated with a particular link is determined.

```
address = segment_address_ (domain, id)
```

where:

domain: `entry_id` input

designates the domain containing the link for the segment. (If null, then the current domain is used.) The link must allow "list" access for the domain.

id: `entry_id` input

designates the entry (within the above domain) for which the corresponding address is to be determined. No special access is required for this operation.

address: pointer result

is assigned the address of the base of the segment referenced by the above link.

GATE: `segment_entry_`

This is an enquiry function that returns the entry id that corresponds to a particular location in the address space of a domain.

```
id = segment_entry_ (domain, address)
```

where:

domain: `entry_id` input

designates the domain containing the link used to reference the segment. (If null, then the current domain is used.) "List" access to the domain must be allowed.

address: pointer input

gives an address within the virtual memory space of the domain. The entry mapped over this location is returned.

id: `entry_id` result

is assigned the id of the entry whose mapping includes the above location. If the address is invalid, then `null_entry_id` is returned and no error is reported.

GATE: `unmap_segment_`

This operation deletes the mapping between an entry and locations within the virtual address space of its containing domain.

`unmap_segment_ (domain, id)`

where:

`domain: entry_id` input

designates the domain containing the entry. (If null, then the current domain is used.) “Modify” access must be permitted.

`id: entry_id` input

designates the segment to be unmapped. This must refer to a segment, but no special access to the target segment is required.

Exceptional conditions:

`syscode.domain_no_modify_access`

is an error signal which occurs if “modify” access to the domain is lacking.

GATE: `list_mapped_segments_`

This entry returns a list of segments mapped in the specified range in the specified domain.

`list_mapped_segments_ (domain, min_addr, max_addr, segment_list_ptr, max_entries)`

where:

`domain`: `entry_id` input

designates the domain whose address space is to be listed. If this is null, the current domain is used. "List" access to the domain is required.

`min_addr`: pointer input

is the lower bound of the portion of the address space to be listed.

`max_addr`: pointer input

is the upper bound of the portion of the address space to be listed.

`segment_list_ptr`: pointer (`mapped_segment_list`) input

gives the address of an area into which a structure giving the list of segments will be placed. The format of the structure appears below.

```

type mapped_segment_list = record
  version: version_id;
  n_entries: unsigned_integer;
  entries: array [1..n_entries] of record
    segment: entry_id;
    address: pointer;
    size: segment_size;
  end;
end;
```

where:

`mapped_segment_list.version`

gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

`mapped_segment_list.n_entries`

gives the number of segments which are mapped in the given range.

`mapped_segment_list.segment`

gives the unique identifier of a mapped segment.

`mapped_segment_list.address`

gives the address of the base of the mapped segment.

`mapped_segment_list.size`

gives the size, in quarterwords, of the address space allocated to the mapped segment.

`max_entries`: `unsigned_integer` input

gives the maximum number of entries that may be returned.

Exceptional conditions:

`mapped_segment_list.n_entries > max_entries`

occurs when there are more mapped segments than have been allowed for. Only the first `max_entries` entries are filled in.

GATE: wire_

This causes a segment or domain to be kept resident in physical memory. (In the case of a segment, the operation applies only if there is an established mapping for the segment within the referencing domain.)

The pages of the object will be brought into main memory and marked so that they will not be evicted by the demand paging mechanism. The subroutine does not return until the operation is complete. This is intended for use with real-time programs, whose code and data must remain resident in main memory to achieve performance requirements.

All records of the segment or domain up to the current maximum length are assumed to exist. If not (i.e. the records have never been written), then the missing records are created as zero pages. Note that truncation of a wired segment does not delete pages of a wired segment but instead zeros them.

This is a privileged operation.

wire_ (id)

where:

id: entry_id input

is an entry for the object to be wired. Some non-null access to the object is required. For segments, any combination of "read", "write", and "execute" permission is sufficient. For domains, any combination of "find", "list", or "modify" is sufficient.

GATE: unwire_

This allows the storage for a segment or domain to be paged. That is, it reverses the effect of the `wire_` primitive. A reference count is kept for each segment or domain which indicates the number of times that it has been wired within different active domains. An object is made unwired only when this reference count is zero. No error is reported if the segment or domain is not currently wired.

This is a privileged operation.

unwire_ (id)

where:

id: entry_id input

is an entry for the object to be unwired. No special access to the object is required.

GATE: `prepage_storage_`

This declares that the task will reference a section of its memory in the near future. A request to bring the addressed area of storage into main memory is queued and processed concurrently with the continued execution of the program.

This provides a way for a performance-critical task to tell the kernel of the order of its future memory accesses, in order that more optimal paging can be achieved.

`prepage_storage_ (segment, start, length)`

where:

`segment: entry_id` input
designates the segment whose pages should be brought into memory. "Read" access to the segment is required.

`start: segment_size` input
gives the offset of the start of the area to be referenced. This offset is rounded down to the nearest record boundary.

`length: segment_size` input
gives the length of the area in quarter-words.

Exceptional conditions:

`syscode.invalid_address`
is an error signal which indicates that the designated area contains non-existent locations.

GATE: `postpurge_storage_`

This declares that the task will not be referencing a section of its memory in the near future. The kernel is then free to evict the storage for the addressed area from main memory.

This provides a way for a performance-critical task to tell the kernel of the order of its future memory accesses, in order that more optimal paging can be achieved.

`postpurge_storage_ (segment, start, length)`

where:

`segment: entry_id` input

designates the segment whose pages should be evicted from memory. "Read" access to the segment is required.

`start: segment_size` input

gives the offset of the start of the area to be purged. This offset is rounded down to the nearest record boundary.

`length: segment_size` input

gives the length of the area in quarter-words.

Exceptional conditions:

`syscode.invalid_address`

is an error signal which indicates that the designated area contains non-existent locations.

GATE: update_storage_

This ensures that a section of the memory of the current task is in a consistent state. Any storage of the addressed area for which a modified version is cached in main memory is written to the permanent storage device. The write occurs concurrently with the continued execution of the program, though the program can wait for the write to complete.

update_storage_ (segment, start, length, wait_flag)

where:

segment: entry_id input

designates the segment whose pages should be updated in permanent storage. "Read" access to the segment is required.

start: segment_size input

gives the offset of the start of the area to be updated. This offset is rounded down to the nearest record boundary.

length: segment_size input

gives the length of the area in quarter-words.

wait_flag: boolean input

indicates that the task should wait until the write completes. In the case where one section of storage must be written before another section is modified, maximum concurrency can be obtained in the following way. Once the changes to the first section are made, an update without wait is requested. Updates to the next section are then computed but not made. Then an update with wait is performed before actually updating the second section. The second update waits only if the first write had not completed.

Exceptional conditions:

syscode.invalid_address

is an error signal which indicates that the designated area contains non-existent locations.

GATE: deactivate_

This gate writes out all modified pages and file maps for a segment, and then removes it from the Active Segment Table. This is a privileged operation.

deactivate_ (segment)

where:

segment: entry_id input

Designates the segment to be deactivated. No particular access to the segment is required.

GATE: `move_data_`

This operation transfers data from a location in one domain to another.

`move_data_ (from_domain, from_address, to_domain, to_address, data_length)`

where:

`from_domain`: `entry_id` input

designates the domain from which information is to be read. (If null, then the current domain is used.) “Use” access to this domain is required.

`from_address`: pointer input

gives the address from which information is to be read. “Read” access must be allowed. (That is, a task executing in the containing domain must be allowed to make a read reference with this pointer.)

`to_domain`: `entry_id` input

designates the domain into which information is to be write. (If null, then the current domain is used.) “Use” access to this domain is required.

`to_address`: pointer input

gives the address to which information is to be written. “Write” access must be allowed.

`data_length`: `segment_size` input

gives the number of quarter-words to be moved. The range of addresses to be read or written must not cross a segment boundary.

Exceptional conditions:

`syscode.invalid_length`

is an error signal which indicates that the “`data_length`” parameter is out of range, i.e. ≤ 0 .

`syscode.invalid_address`

is an error signal. It indicates that some address in the either of the ranges [`address` : `address` + `read_length`) is invalid, or that a range crosses a segment boundary.

GATE: `invoke_domain_`

This makes a copy of a template domain which contains the prelinked environment of a task or a task force, and creates a new domain to hold the copied contents of the template.

Each entry in the template domain is reproduced in the new domain. Names and addresses for the entries are preserved; however, the ids of the new entries differ from the originals.

The entries which reference objects with a “template” property are reproduced by creating a copy of the object with appropriate names and attributes. (Subdomains are never copied, however.) Other entries are reproduced by creating a link to the object. If the entry was originally a link, then the link is duplicated. If the entry was an actual object, the link created uses the original domain’s access to it.

A self-referencing link allowing the new domain full access to itself is also created and entered into the domain under the name “*self*”.

Access to the copied domain is restricted in order that the operation of a protected type manager not be interfered with. The copy is created with an access control list that provides only the parent (the domain containing the copy) with access to the copy, and the modes allowed are limited to that allowed the creator on the template. A special mechanism prevents this modification of the access control list, and hence unauthorized alteration of the copy.

As in the `create_domain_` operation, the owner of the new domain is set to the owner of the domain of execution. The owner can be changed by a subsequent privileged operation.

This gate has not yet been implemented.

`id = invoke_domain_ (template, domain, name, volume)`

where:

template: `entry_id` input

designates the template domain for which an instantiation is to be created. “Invoke” access to this domain is required.

domain: `entry_id` input

designates the domain in which the copy of the template is to be placed. (If null, then the current domain is used.) “Modify” access to this domain is required.

name: `string` input

if non-blank, this gives the name to be given to the new domain. This must not be a duplicate of any other name in the containing domain. If the name is blank, then the domain has no name and can only be referenced by id.

volume: `unique_id` input

designates the volume on which the new domain and the copies of its inferior objects are to be placed. If null, then the default domain volume is taken from the parent domain. If the containing domain resides on a second class volume, this must be the same volume.

id: `entry_id` output

is assigned the id of the newly created domain.

TYPE: `task_state_record`

This structure defines the state of an executing task. Both hardware and software state information is contained in the structure. It is used to denote the initial state of a task, as well as the state of a task during execution.

```

type task_state_record = record
  version: version_id;
  proc_status: longint;
  user_status: longint;
  pc: pointer;
  state_size: integer;
  instruction_state: array [1..64] of integer;
  registers: array [0..31] of integer;
  status: task_status;
  reason: status_code;
  wakeup_mask: boolean;
  priority: task_priority;
  allowed_processors: processor_set;
  quantum: system_time;
  current_domain: entry_id;
end;
```

where:

version

gives the version of the structure being used. The current value is 1. Whenever the structure is passed as an argument, the version must be set to this value. When the structure is an input argument, the version and other fields are set; when used as an output argument, only the version must be set prior to the call.

proc_status

gives the contents of the processor status register.

user_status

gives the contents of the user status register.

pc

gives the location of the instruction currently being executed. When the task is in a waiting state, this gives the address of the instruction at which execution will resume. (That is, the address of the instruction at the return from the wait call.)

state_size, instruction_state

gives hardware dependent information about the state of the current instruction, when its execution has been interrupted. Only `state_size` words of `instruction_state` are in use.

registers

gives the contents of the 32 registers used by the task. It must always be the case that the stack pointer (SP) register points to end of a valid stack.

status

gives the scheduler state of the task. The possible values are: `task_inactive` which means that the task is not allowed to execute and that its state is stored in the storage system. `task_running`, which means that the task is ready to run; `task_stopped`, which means that its execution is suspended; and `task_waiting`, which means that the task is waiting for an event to occur.

reason

gives a reason why the task is in its current state. If the task is stopped, the reason might be that it was stopped by another task, or that the system stopped it because of some error condition. If the `status` field indicates that the task is running, then this field is set to zero.

wakeup_mask

if cleared, then the task is interrupted when it receives a wakeup; otherwise, delivery of the interrupt is deferred until this mask is cleared.

priority

This is the priority at which the task runs. It can be in the range of 0 to *max_priority*.

allowed_processors

This is the set of all processors that this task can run on.

quantum

This is the amount of processor time this task is entitled to before the next task at the same priority level will be run.

current_domain

gives the id of a entry for the "current domain". That is, when a domain parameter is given as a null value, then this value is used instead. Normally, this should be the id of a link for the domain in which the task executes (i.e. the one in which it is catalogued).

TYPE: task_meters_record

This structure defines the various meters maintained by the kernel for an executing task. This information may be used to meter the performance of a program, or to perform resource accounting.

```

type task_meters_record = record
  cpu_time: system_time;
  running_time: system_time;
  preempts: longint;
  waits: longint;
  page_faults: longint;
  segment_faults: longint;
  inst_map_cache_misses: longint;
  data_map_cache_misses: longint;
  inst_cache_misses: longint;
  data_cache_misses: longint;
  instructions_executed: longint;
  flops_executed: longint;
end;

```

where:

cpu_time

gives the amount of processor time used by the task.

running_time

gives the length of time the task has been running. This differs from the cpu time value in the length of time which the task spent waiting for an available processor when it was ready to run. It does not include time during which the task was voluntarily waiting for an event, waiting for page or segment fault processing, or sleeping.

preempts

gives the number of times the task was preempted by higher priority tasks.

waits

gives the number of times the task waited for an event, i.e. the number of wait_ calls.

page_faults

gives the number of pages which were referenced when they were not in main memory.

segment_faults

gives the number of segment faults which were reference when there were no page tables assigned to them.

inst_map_cache_misses, data_map_cache_misses

give the number of virtual addresses which could not be translated in each map cache.

inst_cache_misses, data_cache_misses

gives the number of memory references which could not be satisfied by each cache.

instructions_executed

gives the number of instructions executed by the task.

flops_executed

gives the number of floating point operations executed by the task.

GATE: create_task_

This creates a new task and initializes its execution in the specified domain. It is assumed that all data and program segments required by the task have been pre-initialized.

```
id = create_task_ (domain, name, seal, modes, state)
```

where:

domain: entry_id input

designates the domain in which the task is to be created. This becomes the domain of execution of the task. "Modify" access to this domain is required.

name: string input

if non-blank, this gives the name to be given to the new task. This must not be a duplicate of any other name in the domain. If the name is blank, then the task has no name and can only be referenced by id.

seal: entry_id input

if non-null, designates the task as a protected, extended type object whose representation is a task. Access to unseal the representation is allowed only to this domain (or its agents). No special access to the sealing domain is required.

modes: access_mode_set input

if non-null, this gives the modes of access to be granted to the containing domain. An access-control list entry for the domain is created to permit the access; this entry may be modified or deleted to change the access, subsequently.

state: task_state_record input

gives the initial state of the task. Note that state.pc becomes the starting address of the task.

id: entry_id result

is assigned the id of the newly created task.

Exceptional conditions:

syscode.invalid_address

is an error which is signalled if state.pc is an invalid address in the virtual memory of the domain.

GATE: `get_task_meters_`

This operation returns information about the resources used by the task. If the task is running at the time that this call is made, then the most recent readings for the task are returned. As the task continues to run, the values are immediately obsolete.

`get_task_meters_ (task, meters)`

where:

task: `entry_id` input

designates the task whose meters are to be read. (If null, the meters of the current task are read.)
"Status" access to the task is required.

meters: `task_meters_record` reference input-output

This fields of this structure are set to the values of the various meters for the specified task. (See the definition of the structure.)

GATE: `get_task_state_`

This returns the current state of the task. When the task is running at the time this operation is performed, the state of the task is returned, but is immediately obsolete as the task continues to run.

`get_task_state_ (task, state)`

where:

task: `entry_id` input

designates the task whose state is to be read. (If null, then the state of the current task is obtained.)
"Status" access to the task is required.

state: `task_state_record` reference input-output

The fields of this structure are set on return to give the current state of the task. If `state.status = task_running`, then the information is the most recent available.

GATE: `set_task_state_`

This operation sets the state of a task. As the state includes the program counter, machine register values, and the scheduling status of the task, arbitrary changes in the flow of execution can be effected.

`set_task_state_ (task, state)`

where:

`task: entry_id` input

designates the task whose state is to be altered. (If null, then the current task is used.) "Writestate" access to the task is required. The task must be stopped or inactive

`state: task_state_record` input

gives the new state of the task. In general, the fields of this structure are interpreted as at the time of task creation. Here, however, the `instruction_state` field may have an effect. If non-zero, execution of the instruction address by "pc" is to be resumed. As a consequence, if the point of execution of the task is to be changed, not only must the pc be altered, but `instruction_state` must be set to zero. Note also that the status field may be set to place the task in any of the states: suspended, running, or inactive.

Exceptional conditions:

`syscode.invalid_address`

is an error which is signalled if `state.pc` designates an invalid address.

`syscode.task_not_stopped`

is an error that is signalled if `state.status` is not either *task_inactive* or *task_stopped*.

GATE: `get_task_scheduling_parameters_`

This returns the current scheduling parameters for the task.

```
get_task_scheduling_parameters_ (task, priority, processor_selection, processor_used,  
    quantum)
```

where:

`task`: `entry_id` input

designates the task whose scheduling parameters are to be read. (If null, then the current task is assumed.) "Status" access to the task is required.

`priority`: `task_priority` output

is assigned the current priority of the task.

`processor_selection`: `processor_set` output

is assigned the bit mask indicating which processor the task is allowed to run on. If bit `n` of the mask is on, then the task may run on processor `n`.

`processor_used`: `processor_id` output

is assigned the number of the last processor on which the task was run.

`quantum`: `system_time` output

is assigned the quantum runout value for this task.

GATE: set_task_scheduling_parameters_

This sets the scheduling parameters for a task. This operation can change the “right” of a task to execute on a processor according to the priority scheduling rules, and such changes are effected immediately. This is a privileged operation.

set_task_scheduling_parameters_ (task, priority, processor_selection, quantum)

where:

task: entry_id input

designates the task whose scheduling parameters are to be set. (If null, the link for the current task is used.) “Control” access is required.

priority: task_priority input

gives the priority for the task.

processor_selection: processor_set input

gives a list of processors on which the task is allowed to run. It is given as a bit mask; if bit *n* is on, then the task can run on processor *n*.

quantum: system_time input

gives the run quantum for the task. Every quantum nanoseconds of cpu time, the task will relinquish the processor to another task with the same priority.

GATE: activate_task_

This allows the task to execute. The task is made runnable, and its execution initialized from the stored task state. While the task remains active, the stored state is invalid; should the system halt before the task is deactivated then the state is said to be inconsistent.

activate_task_ (task)

where:

task: entry_id input

designates the task to be activated. (If null, then the current task is assumed.) "Control" access is required.

GATE: deactivate_task_

This prevents a task from executing. This task is halted, and its state stored in permanent storage.

deactivate_task_ (task)

where:

task: entry_id input
designates the task to be deactivated. (If null, then the current task is assumed.) “Control” access is required.

GATE: `suspend_task_`

This operation halts execution of a task. The state of this task becomes `task_stopped`.

`suspend_task_ (task)`

where:

`task: entry_id` input

designates the task to be halted. (If null, the current task is implied.) "Control" access is required.

GATE: `start_task_`

This operation allows a task to execute. It is used to resume execution of a task which is suspended for some reason, and changes the state of the task from suspended, or waiting to running. No error occurs if the task is already in the running state.

`start_task_ (task)`

where:

`task: entry_id` input

designates the task to be started. "Control" access to the task is required.

GATE: kill_task_

This destroys a task.

kill_task_ (task)

where:

task: entry_id input

designates the task to be destroyed. (If null, then the current task is used, i.e., the current task is killing itself.) The link must permit "control" access to the task.

GATE: monitor_task_

This allows a task with appropriate access to monitor state changes in another task. The monitoring task provides a message channel to which the kernel will send messages describing the events. During the activation of a task, there can be only one channel at a time from which a task can be monitored. The monitor function cannot be released, except by deletion of the channel, though the right to monitor the task can be passed by granting access to the monitor channel.

The following events can be monitored by use of this mechanism.

1. Hard-traps and interrupts. The message contains the trap or interrupt information.
2. Creation of tasks. Whenever the monitored task creates another, a message is sent containing a full-access link for the created task.

This is intended to allow a "superior" task to create and manipulate "captive" inferiors.

The message channel mechanism, including this gate, has not yet been implemented.

monitor_task_ (task, monitor_channel)

where:

task: entry_id input

designates the task to be monitored. "Control" access to the task is required.

monitor_channel: entry_id input

gives the channel over which the monitor information is transmitted. Information is transmitted as long as the channel is grabbed by some task; the alternate node of the channel is marked as grabbed by the kernel. The current task must have the right to "transmit" on the channel.

GATE: wakeup_

This sends a wakeup to the task. This interrupt is passed to the user through the soft trap mechanism implemented by the hardware. If a task receives a wakeup while it is running the user interrupt will occur immediately. If the task is blocked the task will be scheduled to run and will receive the interrupt as soon as it starts running. Actually, the interrupt will be delayed if the wakeup mask is set. When the mask is reset any pending wakeup will cause an immediate user interrupt.

wakeup_ (task, wakeup_index)

where:

task: entry_id input

specifies the task which is to be sent the wakeup. (If null, then the current task is assumed.)
"Control" access to this task is required.

wakeup_index: wakeup_id input

gives the number of the wakeup to be sent to the task. When the task is interrupted, the pending wakeup set is delivered to as an interrupt parameter. The bit corresponding to this index, along with any other wakeups not yet delivered, will be set in the mask.

GATE: block_

This suspends the current task until some wakeup is sent to the task. Execution of the task resumes when a wakeup is received, when execution is explicitly restarted by another task (with start_task_), or a maximum wait time is exceeded. The time out is provided for reliability reasons, and ensures that execution is never permanently halted because of a failure of the event to occur. The wakeup mask is reset by this call. If the task is resumed because of a wakeup the accompanying user interrupt will occur before the block subroutine returns.

block_ (time_out)

where:

time_out: system_time input

gives the time until which the task is to wait. This is an absolute time.

GATE: `set_wakeup_mask_`

This operation updates the task's wakeup mask. While the mask is set (true), all wakeups are deferred. If this call changed the mask from true to false and the task has wakeups pending they will be processed (by interrupting the task) before this subroutine returns.

`set_wakeup_mask_ (new_mask, old_mask)`

where:

`new_mask`: boolean input
gives the new value for the task's wakeup mask.

`old_mask`: boolean output
is assigned the mask value previously in effect.

GATE: declare_broadcast_event_

This operation defines a synchronization event which a task is interested in monitoring. When the event is broadcast by some task, the current task will receive a wakeup. (Note that this operation need be performed by the receiver only, not the sender.) The wakeup index is defined by this call.

No error occurs if the task has already declared an event with the specified name (object, event-id). Consequently, this operation may be used to change the wakeup index of a previously defined event.

declare_broadcast_event_ (object, event, wakeup_index)

where:

object: entry_id input

specifies the object with which the event is associated. If this value is null, then the object is taken to be the task itself. The link must allow "listen" access to this object.

event: event_id input

gives the user-assigned id which designates the event.

wakeup_index: wakeup_id input

identifies the wakeup to be sent to the task when the event is broadcast.

Exceptional conditions:

syscode.event_not_declared

is reported if the kernel is unable to enter a declaration of the event for the task. Each installation may place a limit on the number of outstanding events which may be declared by a single task or by all tasks as a group.

GATE: broadcast_

This indicates that a broadcast event has occurred. Any task which has declared its interest in the event is sent a wakeup.

broadcast_ (object, event)

where:

object: entry_id input

specifies the object with which the event is associated. If this value is null, then the object is taken to be the task itself. The link must allow "broadcast" access to the object.

event: event_id input

gives the user defined id of the event.

GATE: delete_broadcast_event_

This deletes a declaration of an event. Its effect is to remove the event from the list of events being monitored by the kernel on behalf of the task. Subsequently, if the event is signalled, no wakeup is sent to the task. No error occurs if the task has not declared the event.

delete_broadcast_event_ (object, event)

where:

object: entry_id input

specifies the object which names the event. If this value is null, then the object is taken to be the task itself. No special access is required to the object.

event: event_id input

gives the user defined id naming the event.

GATE: `set_alarm_`

This causes a wakeup to be sent to the task at some specified time in the future. At any one time, there can be only one alarm for a task. Thus, this operation overrides any previous setting for the same task. If alarms are required at many different times, the program must multiplex the alarm itself.

If the `timer_setting` is `After_Time` this effectively shuts off the alarm.

`set_alarm_ (timer_setting, wakeup_index)`

where:

`timer_setting`: `system_time` input

gives the time at which the interrupt is to occur. This value is an absolute time. If a time in the past is specified, an interrupt occurs immediately.

`wakeup_index`: `wakeup_id` input

designates the wakeup to be sent to the task.

GATE: `set_cpu_alarm_`

This causes a wakeup to be sent to the task after the task has executed for some amount of time. The cpu timer measures only time the processor spent executing the task itself, not any system overhead such as interrupt handling. At any one time, there can be only one cpu time alarm for a task. Thus, this operation overrides any previous setting for the same task. If alarms are required at many different times, the program must multiplex the timer itself.

If the `cpu_timer_setting` is `After_Time` the alarm will be shut off.

`set_cpu_alarm_ (cpu_timer_setting, wakeup_index)`

where:

`cpu_timer_setting`: `system_time` input

gives the time at which the interrupt is to occur. This value is specified as elapsed cpu time since task creation. If the task has already run for the specified time, an interrupt occurs immediately.

`wakeup_index`: `wakeup_id` input

designates the wakeup to be sent when the alarm goes off.

GATE: `set_trap_handlers_`

This is used to set handlers for traps which are handled by the user. These traps include arithmetic errors, wakeup handlers, and kernel error handlers. At the present, this gate is implemented in a fashion specific to the S-1 Mark IIa architecture. This will be changed at some future point.

`set_trap_handlers_ (soft_traps, trpslf_traps)`

where:

`soft_traps`: fault_vector input

is an array of entry variables specifying the procedures to be called for each soft trap. The first element in the trap will be invoked when soft trap 0 occurs, the second element will be invoked when soft trap 1 occurs and so on. See the S-1 Uniprocessor Architecture Manual for a list of soft traps. In addition, Amber defines soft traps 16 through 20 as an error signal trap for errors signalled in the kernel, bad data passed into the kernel, illegal instruction, debugging traps, and wakeups, respectively.

`trpslf_traps`: fault_vector input

is an array of entry variables specifying the procedures to be invoked when the corresponding TRPSLF instruction is executed.

GATE: `get_trap_handlers_`

This is used to retrieve the current vector of handlers for traps which were set by the most recent call to `set_trap_handlers_`

`set_trap_handlers_ (soft_traps, trpslf_traps)`

where:

`soft_traps`: fault_vector reference output

The elements of this array will be filled in with entry variables representing the current handlers for soft traps.

`trpslf_traps`: fault_vector reference output

The elements of this array will be filled in with entry variables representing the procedures to be invoked when the corresponding TRPSLF instruction is executed.

GATE: `add_terminal_`

This configures a terminal on the system. This is a privileged operation.

This gate, like the rest of the terminal interface, is a temporary measure and will be removed once support for message channels and special segments has been implemented.

`add_terminal_ (terminal, owner, wakeup)`

where:

`terminal: terminal_id` input

designates the identifier of the terminal to be configured.

`owner: entry_id` input

designates the owner of the terminal. This task will be sent a wakeup when input is received on this terminal. If this is null, the current task will be made the owner.

`wakeup: wakeup_id` input

specifies which wakeup will be sent when the terminal receives input.

Exceptional conditions:

`syserror.no_device`

There is no physical device with the requested terminal identifier attached to the system.

GATE: `attach_terminal_`

This attaches a terminal to a task. This is a privileged operation.

This gate, like the rest of the terminal interface, is a temporary measure and will be removed once support for message channels and special segments has been implemented.

`attach_terminal_ (terminal, owner)`

where:

terminal: `terminal_id` input
designates the identifier of the terminal to be attached.

owner: `entry_id` input
designates the owner of the terminal. This task will be notified when input is received on this terminal, and can therefore block while waiting for input. If this is null, the current task will be made the owner.

Exceptional conditions:

`syserror.no_device`

There is no physical device with the requested terminal identifier attached to the system.

`syserror.inconsistent_io_operation`

The specified terminal may not be attached because either it is not configured or it is already attached to another task.

GATE: detach_terminal_

This detaches a terminal from a task and returns it to another owner. This is a privileged operation.

This gate, like the rest of the terminal interface, is a temporary measure and will be removed once support for message channels and special segments has been implemented.

detach_terminal_ (terminal, owner, wakeup)

where:

terminal: terminal_id input

designates the identifier of the terminal to be detached.

owner: entry_id input

designates the new owner of the terminal. This task will be sent a wakeup when input is received on this terminal. If this is null, the current task will be made the owner.

wakeup: wakeup_id input

specifies which wakeup will be sent when the terminal receives input.

Exceptional conditions:

syserror.no_device

There is no physical device with the requested terminal identifier attached to the system.

syserror.inconsistent_io_operation

The specified terminal may not be detached because either it is not configured or it is not attached to a task.

GATE: `order_terminal`

This sets terminal characteristics.

This gate, like the rest of the terminal interface, is a temporary measure and will be removed once support for message channels and special segments has been implemented.

`order_terminal_ (terminal, operation, data)`

where:

`terminal: terminal_id` input

designates the identifier of the affected terminal.

`operation: string` input

specifies the operation to be performed. The supported operations are described below.

`data: pointer` input

points to operation-specific data.

The following operations are supported:

`set_break_table`

sets the break table to determine which characters may be echoed by the kernel without notifying the owner during echo negotiation. In this case, *data* points to a 256 element array of boolean values. An element is set to false if the kernel may echo the character with that ascii value, and true if the task should be notified and the character not echoed.

`set_wakeup_id`

Set the wakeup to be sent when the terminal interrupts. *Data* points to a `wakeup_id`.

`start_echo_negotiation`

specifies that the kernel should echo characters on input until a character is entered which is in the break table, or until there is output pending. The daemon makes no guarantees that anything will be echoed. *Data* is ignored.

`stop_echo_negotiation`

The kernel will stop ever echoing input. *Data* is ignored.

Exceptional conditions:

`syserror.unknown_order_operation`

The requested operation is not supported for terminals.

`syserror.no_device`

There is no physical device with the requested terminal identifier attached to the system.

`syserror.inconsistent_io_operation`

The specified terminal may not be operated upon because it is not configured or it is not owned by this task.

GATE: terminal_read_

This is used to read characters from a terminal. If there is no input available, this gate will block waiting for an available character.

This gate, like the rest of the terminal interface, is a temporary measure and will be removed once support for message channels and special segments has been implemented.

terminal_read_ (terminal, buffer, length, echoed)

where:

terminal: terminal_id input

designates the identifier of the terminal to be read from.

buffer: string reference output

is a string buffer where all available input characters will be placed.

length: unsigned_integer output

is set to the number of characters placed in buffer.

echoed: unsigned_integer output

is set to the number of characters placed in buffer which were echoed by the kernel. If echo negotiation is not enabled, this will always be zero.

Exceptional conditions:

syserror.no_device

There is no physical device with the requested terminal identifier attached to the system.

syserror.inconsistent_io_operation

The specified terminal may not be read because it is not configured.

GATE: `terminal_read_char_`

This is used to read a single character from a terminal. If there is no input available, this gate will block waiting for an available character.

This gate, like the rest of the terminal interface, is a temporary measure and will be removed once support for message channels and special segments has been implemented.

```
char = terminal_read_char_ (terminal)
```

where:

terminal: `terminal_id` input
designates the identifier of the terminal to be read from.

char: character output
is the next available character from the terminal.

Exceptional conditions:

`syserror.no_device`

There is no physical device with the requested terminal identifier attached to the system.

`syserror.inconsistent_io_operation`

The specified terminal may not be read because it is not configured.

GATE: `terminal_input_available_`

This is used to determine if any input has been typed on a terminal.

This gate, like the rest of the terminal interface, is a temporary measure and will be removed once support for message channels and special segments has been implemented.

`pending_input = terminal_input_available_ (terminal)`

where:

`terminal:` `terminal_id` input

is the identifier of the terminal to be interrogated about pending input.

`pending_input:` boolean output

is true if there is at least one character available to be read from the terminal and false if there is not.

Exceptional conditions:

`syserror.no_device`

There is no physical device with the requested terminal identifier attached to the system.

`syserror.inconsistent_io_operation`

The specified terminal may not be interrogated because it is not configured.

GATE: `terminal_write_`

This is used to write characters to a terminal.

This gate, like the rest of the terminal interface, is a temporary measure and will be removed once support for message channels and special segments has been implemented.

`terminal_write_ (terminal, buffer, wait)`

where:

`terminal:` `terminal_id` input

designates the identifier of the terminal to be written to.

`buffer:` string input

is a string to be output to the terminal.

`wait:` boolean input

If this flag is set, the gate will not return until the characters have been printed on the terminal. If it is not set, it may return after queueing an output request but before the request is processed.

Exceptional conditions:

`syserror.no_device`

There is no physical device with the requested terminal identifier attached to the system.

`syserror.inconsistent_io_operation`

The specified terminal may not be written because it is not configured.

GATE: `create_channel_`

This operation creates the user and server nodes of a message channel. Each exists as a separate object.

The message channel mechanism, including this gate, has not yet been implemented.

```
create_channel_ (domain, user_name, user_seal, user_modes, user_node_id, server_name,
                 server_seal, server_modes, server_node_id)
```

where:

`domain`: `entry_id` input

designates the domain in which the objects are to be created. "Modify" access to the domain is required.

`user_name`: string input

if non-blank, this gives the name to be given to the new user node object. This must not be a duplicate of any other name in the domain. If the name is blank, then the user node object has no name and can only be referenced by id.

`user_seal`: `entry_id` input

if non-null, designates the user node as a protected, extended type object whose representation is a channel node. Access to unseal the representation is allowed only to this domain (or its agents). No special access is required to the sealing domain.

`user_modes`: `access_mode_set` input

if non-null, this gives the modes of access to the user node object to be granted to the containing domain. An access-control list entry for the domain is created to permit the access; this entry may be modified or deleted to change the access, subsequently.

`user_node_id`: `entry_id` output

is assigned the id of the link for the newly created user node.

`server_name`: string input

if non-blank, this gives the name to be given to the new server node object. This must not be a duplicate of any other name in the domain. If the name is blank, then the server node object has no name and can only be referenced by id.

`server_seal`: `entry_id` input

if non-null, designates the user node as a protected, extended type object whose representation is a channel node. Access to unseal the representation is allowed only to this domain (or its agents). No special access is required to the sealing domain.

`server_modes`: `access_mode_set` input

if non-null, this gives the modes of access to the server node object to be granted to the containing domain. An access-control list entry for the domain is created to permit the access; this entry may be modified or deleted to change the access, subsequently.

`server_node_id`: `entry_id` output

is assigned the id of the link for the newly created server node.

GATE: connect

This is used to establish a connection between the user and server tasks of a message channel. In order for a connection to be completed, tasks must obtain control of the respective nodes via this operation.

If some task has control of the channel, the current task is returned a warning code indicating the condition. In order to forcibly take control of the channel, the current task must use the disconnect operation.

The message channel mechanism, including this gate, has not yet been implemented.

connect_ (channel, open_wakeup, disconnect_wakeup, incoming_packet_wakeup, code)

where:

channel: entry_id input

designates the node which the task is to take control of. "Transmit" access to the channel node is required.

open_wakeup: wakeup_id input

designates the wakeup which is sent when a connection is completed. Since this routine connects this node to this task, the interrupt will occur as soon as the other node is connected by some task. If it is already connected, the wakeup is sent immediately.

disconnect_wakeup: wakeup_id input

designates the wakeup which is sent when the connection is broken. There are four cases: the channel is explicitly disconnected (before or after a connection is completed), one of the nodes of the channel is deleted, one of the controlling tasks is deleted, or the access of one of the controlling tasks is revoked.

incoming_packet_wakeup: wakeup_id input

designates the wakeup which is sent when a packet is sent to the current task before it has asked to receive a packet.

code: status_code output

is assigned a system status code indicating the success or failure of the operation. If set to zero, then the task successfully obtained control of the channel.

Exceptional conditions:

code = syscode.channel_busy

is a warning code which indicates that another task had control of the channel.

code = syscode.cannot_connect_channel

is an error which signifies that the connection could not be created because of internal kernel limitations.

GATE: disconnect_

This resets a channel. Any tasks that have control of a node of a channel (whether or not a connection has been completed) are sent wakeups to notify them of the disconnection. This operation may be performed by any task which has access to use the channel, and may therefore be used to "takeover" the channel if necessary. All outstanding I/O operations are discarded.

The message channel mechanism, including this gate, has not yet been implemented.

disconnect_ (channel)

where:

channel: entry_id input

designates the channel for which control is to be released. "Transmit" access to the channel is required.

GATE: `identify_caller_`

This returns the identity of the task with which the current task is communicating. A connection must exist for a value to be returned.

The message channel mechanism, including this gate, has not yet been implemented.

`identify_caller_ (channel, caller_id)`

where:

`channel: entry_id` input

designates the node of the channel held by the current task. The identify of the task holding the remote node is returned. The current task must have control of the channel.

`caller_id: entry_id` output

is assigned an identifier for the task controlling the remote node of the channel. This a null-access, null-name link for the task. If no connection exists, then null id is returned.

GATE: `send_`

This queues a packet for delivery to the receiving task. If the node specified is the user node, then the packet is sent to the server, and vice versa.

The data in the packet is not actually transmitted until the receiving task explicitly reads the incoming data. At that time, the data is moved directly from the sending to receiving address spaces, and the kernel will issue a wakeup to the sending task.

If the receiving task has not performed a `receive_` operation prior to the `send_`, then a wakeup is sent to the receiving task (when requested) to notify it that a packet is available.

Execution of the task proceeds asynchronously with completion of the transaction. On any one channel, there may be at most one incomplete send operation.

The message channel mechanism, including this gate, has not yet been implemented.

`send_ (node, packet_ptr, packet_length, wakeup_index)`

where:

`node`: `entry_id` input

designates the channel over which the packet is to be transmitted. The task must have control of the channel.

`packet_ptr`: pointer input

gives the address of the start of the buffer containing the message to be sent.

`packet_length`: `segment_size` input

gives the size of the message in quarter-words. This value must be greater than zero.

`wakeup_index`: `wakeup_id` input

designates the wakeup to be sent to the task when the data has been moved out of the buffer or when the receiving task flushes incoming packets

Exceptional conditions:

`syscode.too_many_transactions`

is an error signal which indicates that the limit on outstanding send transactions has been exceeded.

`syscode.zero_length_packet`

is an error which is signalled if packet is of zero or negative length.

GATE: receive

This operation reads the packet from a message channel. If the message node specified is the server node, then packets from the user are read, and vice versa.

The transmission of information into the buffer does not occur until a packet from the sender is available. (The send may occur before or after the receive operation.) Execution of the receiving task continues asynchronously, but when the transfer is completed, an wakeup is issued by the kernel.

On any one channel, at most two receive transactions may be outstanding.

The message channel mechanism, including this gate, has not yet been implemented.

receive_ (node, buffer_ptr, buffer_length, packet_length, wakeup_index)

where:

node: entry_id input

designates the channel from which message data are to be read. The task must have previously grabbed this channel.

buffer_ptr: pointer input

gives the address of a buffer into which the message data are to be read. The buffer should not be accessed until completion of the transaction is signalled.

buffer_length: segment_size input

gives the length of the buffer in quarter-words. If the actual length of the packet to be read is less than the size of the buffer, the buffer is incompletely filled. If the packet is larger, then trailing bytes of the packet are discarded.

packet_length: segment_size reference output

is assigned the length of the packet read. It is set to 0 on return if the packet has not yet arrived, and its address is saved so that it may be updated when the packet arrives.

wakeup_index: wakeup_id input

designates the wakeup to be issued when data has been moved into the buffer.

Exceptional conditions:

syscode.too_many_transactions

is an error signal which indicates that the limit on outstanding send transactions has been exceeded.

GATE: receive_packet_info_

This returns information about the next available incoming packet. The information is available from the time the remote sender issues a send request until the packet is read. As a result, this operation should precede the operation to receive the packet. If this call is issued in advance of a send making a packet available, a status code is returned which indicates that no packet is available.

This operation serves two functions. First, it gives the receiver the length of the incoming packet in advance of reading the actual data. This allows a buffer of appropriate size to be allocated. Second, it can be used to test when a packet is available.

The message channel mechanism, including this gate, has not yet been implemented.

receive_packet_info_ (node, packet_length, code)

where:

node: entry_id input

designates the channel from which the status information is to be returned. The task must have previously grabbed this channel.

packet_length: segment_size output

is assigned the length of the packet which is now available. If there is no packet available, 0 is returned.

code: status_code output

is assigned a system status code indicating the presence of a packet. If set to zero, a packet is available; other codes as enumerated below indicate other conditions.

Exceptional conditions:

code = syscode.channel_empty

is a warning code which indicates that the channel is empty.

GATE: `flush_channel_`

This operation discards packets sent to, but not received by the current task. The sending task is sent an wakeup to notify it that the send buffer is no longer needed. No error occurs if the channel is empty. This may be used by a receiver to delete incoming packets.

The message channel mechanism, including this gate, has not yet been implemented.

`flush_channel_ (channel)`

where:

`channel: entry_id` input

identifies the channel to be flushed. The task must have previously grabbed this channel.

GATE: `create_dummy_object`

This creates a new dummy object. There are no special kernel operations defined for dummy objects; they are used merely as place holders to build user defined objects. As with all other objects, they have a property list which may be used to hold relevant information.

```
id = create_dummy_object (domain, name, seal, modes)
```

where:

domain: `entry_id` input

designates the domain in which the dummy object is to be created. "Modify" access to this domain is required.

name: string input

if non-blank, this gives the name to be given to the new dummy object. This must not be a duplicate of any other name in the domain. If the name is blank, then the dummy object has no name and can only be referenced by id.

seal: `entry_id` input

if non-null, designates the object as a protected, extended type object whose representation is a dummy object. Access to unseal the representation is allowed only to this domain (or its agents). No special access to the sealing domain.

modes: `access_mode_set` input

if non-null, this gives the modes of access to be granted to the containing domain. An access-control list entry for the domain is created to permit the access; this entry may be modified or deleted to change the access, subsequently.

id: `entry_id` result

is assigned the id of the newly created dummy object.

GATE: `set_switch_`

This sets various binary switches associated with the entry.

`set_switch_ (object, switch_name, value)`

where:

object: `entry_id` input

designates the object for which a switch is to be set. The operation is allowed if “put” access to the object is permitted.

switch_name: string input

is the name of the switch to be set. Currently, this must be one of “check_quota”, “damaged”, “trickle”, or “volume_dump”.

value: boolean input

is the new value of the switch.

Exceptional conditions:

`syscode.invalid_switch_name`

This error is signalled when the `switch_name` parameter does not contain one of the valid names mentioned above.

`syscode.not_an_object`

This error occurs when the object named by the `object` parameter is not either a segment or a domain.

GATE: `object_status`

This returns the status information kept for an object.

Where an id is returned as the value of certain fields, such as the creator, it is the identifier of a link entered into the current domain for the express purpose of identifying the referenced object. The link allows no access to the object, and has a null name. The caller must remember to delete the link when finished with the information.

`object_status (object, status)`

where:

`object: entry_id` input

designates the object for which status information is to be returned. The operation is allowed if "get" access to the object is permitted.

`status: object_status_type` reference input-output

is a structure whose fields are assigned the status information. A version field must be set on input and specifies the version of the structure to be used follows:

```

type object_status_type = record
  version: version_id;
  uid: unique_id;
  class: object_class;
  seal: entry_id;
  time_created: system_time;
  creator: entry_id;
  time_entry_modified: system_time;
  time_entry_dumped: system_time;
  volume: unique_id;
  time_used: system_time;
  time_modified: system_time;
  time_dumped: system_time;
  records_used: system_time;
  usage_count: unsigned_integer;
  damaged_flag: boolean;
  max_length: segment_size;
  default_domain_volume: unique_id;
  default_volume: unique_id;
  owner: entry_id;
end;
```

where:

`status.version`

designates the version of the structure being used. Currently, only one version is supported, and the field should be set to 1 prior to the call.

`status.uid`

gives the unique identifier of the object.

`status.class`

gives the primitive class of the object. This field also determines which other fields are used.

`status.seal`

identifies the type of an extended type object. The class field gives the kind of representation of the object. If null, then this is not an extended typed object.

`status.time_created`

gives the time at which the object was created.

status.creator

gives the owner of the domain of the task which created the object. If the access identifier has been subsequently deleted, then a null id is returned.

status.time_entry_modified

gives the last time that the domain information for the object was changed.

status.time_entry_dumped

gives the last time that the status information was dumped by the backup system.

The above fields are defined for all classes of objects. The fields below are only defined for segments and domains.

status.volume

identifies the volume on which the object resides.

status.time_used

gives the last time at which the contents of the segment or domain were accessed.

status.time_modified

gives the last time at which the segment or domain subtree was last modified.

status.time_dumped

gives the last time that the contents of the segment or domain were dumped by the backup system.

status.records_used

gives the number of records of storage used by the object.

status.usage_count

gives the number of times the contents of the segment or domain were accessed. This counts the number of times a link for the object was obtained — with sufficient access to use the contents of the object.

status.damaged_flag

if true, then the storage system found the object in an inconsistent state.

status.max_length

gives the maximum length of the contents of the object in quarter-words. The value is enforced to the next highest page boundary.

The following fields are defined only for domain class objects.

status.default_domain_volume

identifies the volume on which inferior domains will be created by default.

status.default_volume

identifies the volume on which inferior, non-domain branches will be created by default.

status.owner

identifies the owner of the domain.

GATE: `put_property_`

This operation sets the value of a property on the property list of an object. If there is already a property with the same name, the value is changed; otherwise a new property is created and initialized.

`put_property_ (object, name, value)`

where:

object: `entry_id` input

designates the object whose property list is to be changed. The operation is allowed if “put” access to the object is allowed.

name: `string` input

gives the name of the property to be set. (This must conform to the rules for object names.)

value: `string` input

gives the new value for the property.

GATE: `get_property_`

This operation obtains the value of a property on the property list of an object. If there exists a property with the specified name, then its value is returned; if there is no such property, then it is assumed to have a value of `""`.

`get_property_ (object, name, value, value_length)`

where:

object: `entry_id` input

designates the object whose property list is to be examined. The operation is allowed if “get” access to the object is permitted.

name: string input

gives the name of the property to be obtained. This name must conform to the rules for object names.

value: string output reference

is assigned the current value of the property. If the maximum length of the variable is less than the length of the value of the property, then the return value is truncated.

value_length: `char_index` output

is set to the length of the value. Zero is used for the length of an undefined property.

Exceptional conditions:

length (value) < value_length

occurs when the string allocated to hold the value of the property is too short. The value of the property is truncated.

GATE: `put_property_test_`

This updates the value of a property conditional on its present value. The operation is indivisible with respect to other property list operations to allow manipulation of “lock” values. (As with the `get_property_` operation, if the property is not already defined, its current value is taken as the null string.)

flag = `put_property_test_ (object, name, test_value, new_value)`

where:

object: `entry_id` input

designates the object whose property list is to be updated. The operation is allowed if both “get” and “put” access to the object is permitted.

name: `string` input

gives the name of the property to be updated. The name must conform to the rules for object names.

test_value: `string` input

gives the value to be compared with the present value of the property. The value is altered only if this matches the present value; note that “” matches an undefined property.

new_value: `string` input

gives the value to be assigned to the property.

flag: `boolean` result

is set true if `test_value` equals the current value of the string; false, otherwise.

GATE: `delete_property_`

This removes a property from the property list of an object. No error occurs if no property with the specified name exists.

`delete_property_ (object, name)`

where:

`object. entry_id` input

designates the object whose property list is to be modified. The operation is allowed if “put” access to the object is allowed.

`name`: string input

gives the name of the property to be deleted.

GATE: `delete_property_test_`

This deletes a property conditional on its present value. The operation is indivisible with respect to other property list operations.

flag = `delete_property_test_` (**object**, **name**, **test_value**)

where:

object: `entry_id` input

designates the object whose property list is to be updated. The operation is allowed if both “get” and “put” access to the object is permitted.

name: `string` input

gives the name of the property to be updated. The name must conform to the rules for object names.

test_value: `string` input

gives the value to be compared with the present value of the property. The value is altered only if this matches the present value; note that “” matches an undefined property.

flag: boolean result

is set true if `test_value` equals the current value of the string; false, otherwise.

GATE: `test_property_`

This operation tests for the presence of a particular property on the property list of an object.

`flag = test_property_ (object, name, value_length)`

where:

`object: entry_id` input

designates the object whose names are to be changed. The operation is allowed if “get” access to the object is allowed.

`name: string` input

gives the name of the property whose presence is to be tested

`value_length: char_index` output

is set to the length of the value of the tested property. Zero is returned for the length of an undefined property.

`flag: boolean` result

is set to one if there is a property with the specified name; false, otherwise.

GATE: `list_properties`

This operation lists the properties of an object. The names and lengths of the property values are returned; the values themselves must be obtained by individual calls.

`list_properties (object, property_list_ptr, max_properties)`

where:

`object`: `entry_id` input

designates the object whose names are to be changed. The operation is allowed if “get” access to the object is allowed.

`property_list_ptr`: pointer (`property_list`) input

gives the address of an area into which a structure giving the list of properties will be placed. The format of the structure appears below.

```

type property_list = record
    version: version_id;
    n_properties: unsigned_integer;
    list: array [1..n_properties] of record
        name: object_name;
        property_value_length: unsigned_integer;
    end;
end;
```

where:

`property_list.version`

gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

`property_list.n_properties`

gives the number of properties which are defined.

`property_list.name`

gives the name of a property.

`property_list.property_value_length`

gives the current length of the string value of the property.

`max_properties`: `unsigned_integer` input

gives the maximum number of properties that may be returned.

Exceptional conditions:

`property_list.n_properties > max_properties`

occurs when there are more properties than have been allowed for. Only the first `max_properties` property entries are filled in.

GATE: `set_default_volume_`

This sets the default volume on which segment objects in a specified domain are to be placed. When a domain is initially created, the default is set to the volume on which the domain, itself, resides.

The quota mechanism, including this gate, has not yet been implemented. The entire hierarchy is constrained to fit on a single volume.

`set_default_volume_ (domain, volume)`

where:

domain: `entry_id` input

designates the domain for which the default is to be set. The default applies to branch objects created within this domain. "Modify" access to this domain is required.

volume: `unique_id` input

identifies the volume to be used. No special access is required. If the domain resides on a second class volume, then this must be the same volume.

GATE: `set_default_volume_domains_`

This sets the default volume on which sub-domain objects are to be placed. When a domain is initially created, the default is set to the volume on which the domain, itself, resides.

The quota mechanism, including this gate, has not yet been implemented. The entire hierarchy is constrained to fit on a single volume.

```
set_default_volume_domains_ (domain, volume)
```

where:

`domain`: `entry_id` input

designates the domain for which the default is to be set. The default applies to domain objects created within this domain. "Modify" access to this domain is required.

`volume`: `unique_id` input

identifies the volume to be used. No special access is required. If the domain resides on a second class volume, then this must be the same volume.

GATE: create_quota_account_

This creates a quota account for a volume, or alters the parameters of an existing account.

In effect, this enables the creation of domains and segments on that volume within the specified domain subtree; in addition, it places an absolute limit on the amount of storage on the volume which can be used. If the account already exists, the effect of this operation is to alter the limit.

The volume is identified by a unique identifier, generated and cataloged external to the normal kernel mechanisms. Thus, use of this operation must be made privileged in order to avoid unauthorized use of storage on the volume.

The quota mechanism, including this gate, has not yet been implemented.

create_quota_account_ (domain, volume, quota)

where:

domain: entry_id input

designates the top of the domain subtree to which the quota account applies. The subtree includes all inferior domains that do not have their own separate account; it does not include this domain itself, its usage is billed to a higher account. No special access is required to this domain.

volume: unique_id input

designates the volume on which the quota may be used.

quota: unsigned_integer input

gives the maximum number of physical storage records which can be by objects in the subtree which reside on the volume. If the account already exists, this gives the new maximum. It need not be greater or less than the existing maximum or the number of records currently being used.

GATE: delete_quota_account

This deletes a quota account. In effect, it prohibits the creation of domains and segments on that volume within the specified domain subtree (unless there is a quota account for the same volume higher in the domain hierarchy.) This operation is not allowed if there is a non-zero usage on the account (i.e. if there still exist any objects residing on the volume).

The volume is identified by a unique identifier, generated and cataloged external to the normal kernel mechanisms. Thus, use of this operation must be made privileged in order to avoid unauthorized use of storage on the volume.

The quota mechanism, including this gate, has not yet been implemented.

delete_quota_account (domain, volume)

where:

domain: entry_id input

designates the top of the domain subtree to which the quota account applies. No special access is required to this domain.

volume: unique_id input

designates the volume on which the quota may be used. "Allocate" access for this volume is required.

Exceptional conditions:

syscode.volume_in_use

is an error signal which occurs when an attempt is made to delete the allocation, and there remain objects (even with zero quota usage) residing on the volume.

GATE: `set_quota_limit_`

This sets a optional limit on the use of storage on a volume. The total number of records of physical storage used by objects within the domain (or its sub-domains) cannot exceed this limit. (Note that this limit does not apply to sub-domains that have a separate quota account for the volume.) If a sub-domain has its own quota limit, consumption of storage by objects within the sub-domain is checked against its own limit and that of any superiors.

In contrast to the creation of a quota account which is a privileged system function, this operation may be performed by an unprivileged user to control use of the physical storage by objects for which he is responsible.

The quota mechanism, including this gate, has not yet been implemented.

`set_quota_limit_ (domain, volume, quota)`

where:

domain: `entry_id` input

designates the domain on which the quota is to be applied. "Modify" access to this domain is required.

volume: `unique_id` input

designates the volume for which the quota limit applies. No special access is required to this volume; however, a quota account for the volume must have been created in the domain or one of its superiors.

quota: integer input

is the quota value to set. A value of -1 denotes no limitation.

Exceptional conditions:

`syscode.no_quota_account`

is an error signal which occurs if there is no quota account for the volume which applies to the subtree containing the domain.

GATE: `list_quota_`

This is an enquiry operation that provides quota information about a domain. For each volume, the number of records charged against the domain's quota, and the quota allocation and authorization limits are returned.

The quota mechanism, including this gate, has not yet been implemented.

`list_quota_ (domain, quota_list_ptr, max_volumes)`

where:

`domain: entry_id` input

designates the domain whose quota information is to be returned. "List" access to the domain is required.

`quota_list_ptr: pointer (quota_list)` input

gives the address of an area into which a structure giving the list of per-volume quota information can be place. The format of this structure is as follows.

```

type quota_list = record
  version: version_id;
  n_volumes: unsigned_integer;
  info: array [1..n_volumes] of record
    volume: unique_id;
    quota_account_limit: integer;
    quota_limit: integer;
    usage: unsigned_integer;
    time_record_product: longint;
  end;
end;
```

where:

`quota_list.version`

gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

`quota_list.n_volumes`

is the number of volumes for which quota information is kept. This includes volumes for which this domain has an allocation, and volumes for which containing domains have allocations.

`quota_list.volume`

gives the id of the volume for which the quota information applies. This link serves only to identify the volume, and grants no access to it.

`quota_list.quota_account_limit`

gives the limit for a quota account associated with this domain. If there is no account, then a value of -1 is returned. (There must exist a quota account in a superior domain.)

`quota_list.quota_limit`

gives the number of records allocated for use in this domain. If there is no limit, a value of -1 is assigned.

`quota_list.usage`

gives the number of records on the volume charged to this domain. The value includes the records of segments and domains in the current domain, as well as the individual usages of the subdomains.

`quota_list.time_record_product`

gives the integral of record usage over time charged against this domain.

`max_volumes`: integer input

gives the maximum number of volume quota information entries that may be returned.

Exceptional conditions:

`quota_list.n_volumes > max_volumes`

occurs if there are more volume entries than there is space allocated for. The available entries are filled in.

GATE: `salvage_domain`

This invokes the domain salvager to check the integrity of an Amber domain, rebuilding it if necessary. This can optionally be used to reclaim lost space in the directory area and to delete connection failures. This is a privileged operation.

```
severity = salvage_domain (domain, options, copy_ptr, copy_len)
```

where:

`domain`: `entry_id` input

is the domain to be salvaged. "Modify" access to the domain is required.

`options`: `dir_check_options` input

is a structure containing the various options available. It is described as follows:

```
type dir_check_options = packed record
  delete_connection_failures: boolean;
  rebuild: boolean;
  check_vtoce: boolean;
  copy: boolean;
end;
```

where:

`dir_check_options.delete_connection_failures`

This is set if the salvager should read the VTOC entries for each branch in the domain and delete the entries from the domain if the VTOCE does not match the domain entry.

`dir_check_options.rebuild`

This is set if the domain should be reconstructed even if there are no errors. Normally, the domain will be rebuilt only if a severe error is found. This option allows reclamation of wasted space in the domain area and computation of more suitable hash table sizes.

`dir_check_options.check_vtoce`

If this is set, VTOCE's for all branches are read, and errors are reported if the VTOCE attributes do not match those in the entry. Connection failures are not deleted. If `delete_connection_failures` is on, this option is redundant.

`dir_check_options.copy`

If this is set, and if the domain must be rebuilt, a copy of the domain will be made before the rebuild is done. A non-nil pointer must be supplied in the `copy_ptr` parameter.

`copy_ptr`: pointer input

This is a pointer to an area where a copy of the domain may be placed before rebuilding it.

`copy_len`: segment size output

If a copy is made, this is set to the size, in quarterwords, of the damaged domain that was copied to the area pointed to by `copy_ptr`

`severity`: `dir_severity` result

This set to one of the enumerated type `dir_severity`, which contains the elements *info*, *fix*, *rebuild*, and *failure*. *INFO* denotes no problems, but some informational messages may have been placed in the syserr log. *Fix* means minor, correctable errors occurred. *Rebuild* means that either the rebuild option was specified, or severe errors were detected. *Failure* means that the salvager was unable to construct a correct copy.

GATE: `priv_delete_`

This is used to remove a domain or segment from the storage system if its corresponding entry in a disk volume table of contents is missing. This removes the directory entry for the specified object only. Unlike `delete_entry_`, it will not delete inferiors to a domain. It should only be called when `delete_entry_` fails. This is a privileged operation.

`priv_delete_ (domain, id)`

where:

domain: `entry_id` input

Designates the domain containing the entry to be deleted. "Modify" access to the domain is required.

id: `entry_id` input

Designates the entry that is to be deleted.

GATE: `adopt_segment`

This creates a directory entry for an existing “orphan” entry in a disk table of contents. It is a privileged operation.

```
uid = adopt_segment (domain, name, volume, sspart, vtocx, modes)
```

where:

`domain`: `entry_id` input

Designates the domain where the adopted segment should be placed. “Modify” access to the domain is required.

`name`: `object_name` input

If this is non-blank, it is the name to be given to the adopted segment. If it is blank, then the segment has no name and must be referenced by id.

`volume`: `unique_id` input

Designates the storage system volume containing the orphan entry.

`sspart`: `unique_id` input

Designates the storage system partition containig the orphan entry.

`vtocx`: `unsigned_integer` input

Designates the position in the volume table of contents of the orphan entry.

`modes`: `access_mode_set` input

If this is non-null, it designates the initial access to be granted to the creating domain.

`uid`: `entry_id` result

This is the unique identifier of the adopted entry as obtained from its entry in the volume table of contents.

GATE: `sweep_partition`

This is used to traverse a storage system partition and search for entries in the partition's table of contents which are not in the storage system hierarchy. Such entries may optionally be adopted or deleted. A message is placed in the `syserr` log for each orphan entry detected. This is a privileged operation.

`sweep_partition (sspart, domain, adopt, delete)`

where:

`sspart`: `partition_id` input

designates the storage system partition to be examined.

`domain`: `entry_id` input

designates a domain into which any orphan entries detected by the sweep will be adopted. If the `adopt` parameter is not set, this parameter is ignored. If null, the current domain is used. "Modify" access to this domain is required.

`adopt`: boolean input

If this parameter is set, orphan entries will be adopted into the hierarchy.

`delete`: boolean input

If this parameter is set, orphan entries will be deleted from the partition's table of contents.

GATE: `unique_id`.

This generates and returns a unique identifier.

```
uid = unique_id ()
```

where:

uid: `unique_id` result
is the unique value generated.

GATE: `clock_read_`

This returns the current reading of the system calendar clock.

```
time = clock_read_ ()
```

where:

time: `system_time` result
is assigned the current clock reading.

GATE: `clock_set`

This sets the value of the system calendar clock. This is a privileged operation.

`clock_set (now)`

where:

`now`: `system_time` input

gives the date and time to which the clock is to be set.

GATE: `add_processor_`

This operations declares that a processor is ready to run tasks. This is a privileged operation.

The dynamic reconfiguration mechanism, including this gate, has not yet been implemented.

`add_processor_ (processor no)`

where:

`processor_no: processor_id` input
gives the identifier of the processor to be added to the system configuration.

Exceptional conditions:

`syscode.invalid_processor_no`
is an error which occurs when a bad processor number is used.

`syscode.processor_online`
is an error signal which occurs when an attempt is made to add a processor which is already in the system configuration.

GATE: delete_processor_

This operation causes a processor to be deleted. All tasks operating on the processor are transferred to other processors if their processor selection set so allows; tasks dedicated to the particular processor are stopped. This is a privileged operation.

The dynamic reconfiguration mechanism, including this gate, has not yet been implemented.

delete_processor_ (processor_no)

where:

processor_no: processor_id input
gives the identifier of the processor to be removed from the system configuration.

Exceptional conditions:

syscode.invalid_processor_no
is an error which occurs when a bad processor number is used.

syscode.processor_offline
is an error which occurs when an attempt is made to delete a processor which is not currently in the system configuration.

GATE: `add_memory_`

This operation declares that a region of physical memory is available to be used by the system. The region specified may be less than a complete memory unit. This is a privileged operation.

The dynamic reconfiguration mechanism, including this gate, has not yet been implemented.

`add_memory_ (address, length)`

where:

address: `physical_address` input
gives the physical starting address of the region of memory to be added.

length: `physical_address` input
gives the length of the region of memory to be added.

Exceptional conditions:

`syscode.invalid_memory_length`
is an error which is signalled when the length parameter is out of range, i.e. $\text{length} \leq 0$.

`syscode.memory_online`
is an error which is signalled when some part of the region specified is already in the system configuration. When this error occurs, no alteration to the configuration occurs.

GATE: delete_memory_

This operation removes a region of physical memory from use. The region may be less than the size of an actual memory unit. Any wired pages in this region are transferred to unaffected memory. This is a privileged operation.

The dynamic reconfiguration mechanism, including this gate, has not yet been implemented.

delete_memory_ (address, length)

where:

address: physical_address input
gives the physical starting address of the region of memory to be deleted.

length: physical_address input
gives the length of the region of memory to be deleted.

Exceptional conditions:

syscode.invalid_memory_length
is an error which is signalled when the length parameter is out of range, i.e. $\text{length} \leq 0$.

syscode.memory_offline
is an error which is signalled when some part of the region specified is not currently in the system configuration. When this error occurs, no alteration to the configuration occurs.

GATE: shutdown_

This is used to shut down the Amber operating system. This gate does not return. It activates a kernel task which is charged with stopping the traffic controller, cleanly shutting down the file system, and quiescing the storage system devices. This is a privileged operation.

shutdown_ ()

GATE: `mount_volume_`

This is used to mount a storage system volume. This is a privileged operation.

`mount_volume_ (volume, drive)`

where:

`volume`: `unique_id` input

This is the unique identifier of the storage system volume to be mounted.

`drive`: `unique_id` input

This is the unique identifier of the physical device (e.g. disk drive) where the volume should be mounted.

Exceptional conditions:

`syserror.volume_already_mounted`

The requested volume is already mounted.

`syserror.drive_in_use`

There is already a volume mounted on the requested device.

`syserror.bad_disk_header`

The volume on the requested drive is not an Amber storage system volume.

GATE: `dismount_volume`

This is used to dismount a storage system volume. This is a privileged operation.

`dismount_volume (volume)`

where:

`volume`: `unique_id` input

This is the unique identifier of the storage system volume to be dismounted.

Exceptional conditions:

`syscode.invalid_volume`

The requested volume is not mounted.

`syscode.volume_in_use`

The volume may not be dismounted because it contains active storage system partitions.

GATE: `mount_partition_`

This is used to mount a storage system partition. The volume containing the partition must already be mounted. This is a privileged operation.

`mount_partition_ (partition)`

where:

`partition`: `unique_id` input
Designates the partition to be mounted.

Exceptional conditions:

`syserror.volume_not_mounted`
No volume containing the requested partition is mounted.

GATE: `dismount_partition_`

This is used to dismount a storage system partition. This is a privileged operation.

`dismount_partition_ (partition)`

where:

`partition`: `unique_id` input

Designates the partition to be dismounted.

Exceptional conditions:

`syscode.sspart_not_mounted`

The requested storage system partition was not mounted.

GATE: `list_volumes`

This is used to obtain a list of all storage system volumes currently mounted.

```
list_volumes_ (volume_list, max_entries)
```

where:

`volume_list`: pointer (`volume_list`) input
is the address of an area into which the volume list is to be placed. The format is given below.

```
type volume_list = record
  version: version_id;
  n_volumes: unsigned_integer;
  volume: array [1..n_volumes] of record
    drive: unique_id;
    volume_header: disk_header;
  end;
```

where:

`volume_list.version`
gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

`volume_list.n_volumes`
gives the number of entries in the mounted volume list.

`volume_list.drive`
is the unique identifier of the drive the volume is mounted on.

`volume_list.volume_header`
is the disk header for the volume.

`max_entries`: integer input
is the maximum number of volume list entries that may be returned.

Exceptional conditions:

`volume_list.n_volumes > max_entries`
occurs if there are more mounted volumes than space has been allocated for. As many entries as possible are filled in.

GATE: `list_partitions_`

This is used to obtain a list of all storage system partitions currently mounted.

```
list_partitions_ (partition_list, max_entries)
```

where:

`partition_list`: pointer (`partition_list`) input
is the address of an area into which the partition list is to be placed. The format is given below.

```
type partition_list = record
    version: version_id;
    n_partitions: unsigned_integer;
    partition: array [1..n_partitions] of record
        partition_header: partition_header;
    end;
```

where:

`partition_list.version`
gives the version of the structure being used. Currently, only one version is supported, and this field should be set to 1 prior to the call.

`partition_list.n_partitions`
gives the number of entries in the mounted partition list.

`partition_list.partition_header`
is the disk header for the partition.

`max_entries`: integer input
is the maximum number of volume list entries that may be returned.

Exceptional conditions:

`volume_list.n_partitions > max_entries`
occurs if there are more mounted partitions than space has been allocated for. As many entries as possible are filled in.

GATE: `salvage_volume_`

This is used to salvage a storage system volume's volume header. This is a privileged operation.

`salvage_volume_ (volume, drive_id)`

where:

`volume`: `unique_id` input

This is the unique identifier of the storage system volume to be salvaged. The volume must not be mounted.

`drive`: `unique_id` input

This is the unique identifier of the physical device (e.g. disk drive) where the volume can be found.

GATE: `salvage_partition_`

This is used to salvage a storage system partition. This is a privileged operation.

`salvage_partition_ (partition)`

where:

partition: `unique_id` input

This is the unique identifier of the storage system partition to be salvaged. The partition must not be mounted.

GATE: format_volume_

This is used to format a volume as an Amber storage system volume. This is a privileged operation.

volume := format_volume_ (drive, name, device, serial_no, comment, size)

where:

drive: unique_id input

designates the physical device where the unformatted volume may be found.

name: string input

is the name to be assigned to the new volume.

device: string input

is the device name to be assigned to the new volume.

serial_no: string input

is the serial number to be assigned to the new volume.

comment: string input

is a comment string to be associated with the new volume.

size: unsigned_integer input

is the size of the new volume in quarterwords. This includes the space taken up by the volume header.

volume: unique_id output

is the unique identifier assigned to the new volume.

Exceptional conditions:

syscode.volume_in_use

The volume mounted on the specified drive is in use.

GATE: `format_partition_`

This is used to format a partition as an Amber storage system partition. This is a privileged operation.

```
partition := format_partition_ (volume, name, comment, start, length, vtoc_size, vsmt_size,  
    second_class, connect_checking)
```

where:

volume: `unique_id` input

designates the volume where the partition should be formatted. The volume must be mounted.

name: `string` input

is the name to be assigned to the new partition.

comment: `string` input

is a comment string to be associated with the new partition.

start: `unsigned_integer` input

is the starting address of the partition within the volume. The address is specified in quarterwords.

length: `unsigned_integer` input

is the size of the partition in quarterwords. This includes the header, VTOC, and VSMT.

vtoc_size: `unsigned_integer` input

is the number of entries to be allocated in the volume table of contents. (VTOC)

vsmt_size: `unsigned_integer` input

is the number of entries to be allocated in the volume segment map table. (VSMT)

second_class: `boolean` input

If set, this designates this partition as a second-class partition.

connect_checking: `boolean` input

If set, this specifies that record connect checking is to be performed on this partition. This increases storage system reliability at the expense of disk space

partition: `unique_id` output

is the unique identifier assigned to the new partition.

Exceptional conditions:

`syscode.invalid_volume`

The specified volume is not mounted.

GATE: `delete_partition_`

This is used to delete a storage system partition from a storage system volume. The partition must not be mounted, but its containing volume must be. This is a privileged operation.

`delete_partition_ (partition)`

where:

`partition`: `unique_id` input
designates the partition to be deleted.

Exceptional conditions:

`syserror.volume_not_mounted`
No volume containing the specified partition is mounted.

GATE: `read_vtoce_`

This is used to read an entry from the volume table of contents. This is a privileged operation.

`read_vtoce_ (partition, vtocx, vtoce)`

where:

`partition`: `unique_id` input

is the unique identifier of the storage system partition containing the VTOCE to be read.

`vtocx`: `unsigned_integer` input

is the index of the desired entry in the volume table of contents.

`vtoce`: `vtoce` reference output

The fields of this structure are filled in with the contents of the requested volume table of contents entry. See ADN-?? for a description of the VTOCE structure.

Exceptional conditions:

`syscode.sspart_not_mounted`

The requested storage system partition is not mounted.

GATE: `read_vsmte_`

This is used to read an entry from the volume segment map table. This is a privileged operation.

`read_vsmte_ (partition, vsmtx, vsmte)`

where:

`partition`: `unique_id` input

is the unique identifier of the storage system partition containing the VSMTE to be read.

`vsmtx`: `unsigned_integer` input

is the index of the desired entry in the volume segment map table.

`vsmte`: `vsmte` reference output

The fields of this structure are filled in with the contents of the requested volume segment map table entry. See ADN-?? for a description of the VSMTE structure.

Exceptional conditions:

`syscode.sspart_not_mounted`

The requested storage system partition is not mounted.

GATE: `delete_vtoce_`

This is used to delete an entry from the volume table of contents. This operation only deletes the vtoce; the partition salvager must be invoked to delete any vsmtes and disk records owned by this vtoce. This is a privileged operation.

`delete_vtoce_ (partition, vtocx)`

where:

`partition`: `unique_id` input
is the unique identifier of the storage system partition containing the VTOCE to be deleted.

`vtocx`: `unsigned_integer` input
is the index of the desired entry in the volume table of contents.

GATE: `get_kernel_address_`

This is used to obtain the address within the kernel address space of any non-filesystem segment mapped in the kernel address space. This is a debugging tool and its use is a privileged operation.

`address := get_kernel_address_ (name)`

where:

name: string input

is the 8 character name of the kernel segment.

address: pointer output

is the address within the kernel address space where the specified segment is mapped. If the segment is not mapped, nil is returned. This pointer is intended to be passed as a parameter to the `copy_out_` gate and should not be referenced through.

GATE: `copy_out_`

This is used to copy data from the kernel address space into a user's address space. This is used for debugging and is a privileged operation.

`copy_out_ (kernel_address, user_address, length)`

where:

`kernel_address`: pointer input

is a pointer to the data in the kernel which is to be copied into the caller's address space.

`user_address`: pointer input

is a pointer to an area where the kernel data should be placed.

`length`: `segment_size` input

is the length, in quarterwords, of the area to be copied.

GATE: `copy_syserr_log_`

This is used to copy out the system error log from the kernel buffer into the file system. This must be done periodically because the kernel paged syserr log is of a fixed size. This is a privileged operation.

`copy_syserr_log_ (log_buffer, length)`

where:

`log_buffer`: string reference output
is a buffer where the paged syserr log should be placed.

`length`: `unsigned_integer` output
is the length, in quarter-words, of the log which was placed in log buffer.

Common Exceptions

There are a number of error conditions which are detected by many kernel operations. For example, errors resulting from incorrect access or invalid names. These common errors have been omitted from the specification of each interface; only those which are peculiar to a particular operation, or of special significance to the operation are given in specification.

The following is a list of the error codes commonly returned by the kernel operations. This codes are passed to the user as a parameter to the *error* exception.

`syscode.entry_not_found`

An entry (object or link) having the specified id cannot be found.

`syscode.not_a_domain`

The operation requires a domain object.

`syscode.not_a_segment`

The operation requires a segment object.

`syscode.not_a_task`

The operation requires a task object.

`syscode.not_a_channel`

The operation requires a message channel object.

`syscode.object_no_get_access`

"Get" access to an object is not allowed.

`syscode.object_no_put_access`

"Put" access to an object is not allowed.

`syscode.object_no_listen_access`

"Listen" access to an object is not allowed.

`syscode.object_no_broadcast_access`

"Broadcast" access to an object is not allowed.

`syscode.domain_no_find_access`

"Find" access to a domain is not allowed.

`syscode.domain_no_list_access`

"List" access to a domain is not allowed.

`syscode.domain_no_modify_access`

"Modify" access to a domain is not allowed. Modify access is required to create objects and to change their names, access control lists or attributes.

`syscode.domain_no_use_access`

"Use" access to a domain is not allowed.

`syscode.domain_no_invoke_access`

"Invoke" access to a domain is not allowed.

`syscode.seg_no_read_access`

"Read" access to a segment is not allowed.

`syscode.seg_no_write_access`

"Write" access to a segment is not allowed.

`syscode.task_no_status_access`

"Task" access to a task is not allowed.

syscode.task_no_writestate_access

“Writestate” access to a task is not allowed.

syscode.task_no_control_access

“Control” access to a task is not allowed.

syscode.channel_no_transmit_access

“Transmit” to a message channel is not allowed.

syscode.domain_full

A domain cannot be modified because space cannot be allocated in the domain for some internal kernel structure.

syscode.invalid_volume

The volume specified in the creation of a segment or domain is not currently mounted on the system.

syscode.incorrect_volume

The volume specified in the creation of a segment or domain was not the same as the second class volume on which the containing domain resides.

syscode.insufficient_quota

A segment or domain cannot be created because a quota limit has been exceeded. The limit may be for either the volume holding the containing domain or the volume on which the object is to be placed. Contrast this with **syscode.no_quota**

syscode.no_quota

A segment or domain cannot be created because there is no quota account for the volume within the containing domain or one of its parents. This occurs when use of the volume has not been authorized.

syscode.bad_object_name

An object name is too long or contains invalid characters. Object names must be 48 characters or less and use only the 95 character printing ASCII set.

syscode.bad_property_name

A property name is too long or contains invalid characters.

syscode.name_duplication

An entry cannot be created because the name to be given to it conflicts with that of an existing entry. When the name is given as all blanks, then the object will have no name and this error cannot occur.

syscode.object_not_created

A object cannot be created for some reason. This is a catch-all code.

syscode.invalid_version

The version specified for a structure is not a current version.

syscode.invalid_access_path

A link's access path is invalid. The error when (1) a path id refers to an object rather than a link, (2) if the first component of the path cannot be legally used by the domain creating or holding the link.

syscode.not_an_object

The operation applies only to object entries, not to link entries as well. For example, the **set_acl** operation is applicable only to objects.

syscode.object_not_addressible

A mapping operation can be applied only to segment entries (including links to segments).

syscode.task_inactive

The operation (start, suspend, etc.) applies only to active tasks.

syscode.invalid_wakeup_id

A wakeup index is null or exceeds the maximum allowed by the installation.

`syscode.user/server_node_deleted`

A message channel operation cannot be performed because one node of the channel has been deleted.

`syscode.channel_disconnected`

**A message channel operation cannot be performed because the current task does not have a connection.
Either the task does not have control of the channel, or no connection exists on the channel.**

Gate Index

create_domain_	19
set_domain_owner_	20
move_object_	21
lookup_	22
list_entries_	23
list_entries_status_	24
list_entries_by_name_	26
link_info_	28
get_pathname_	30
list_names_	32
set_names_	33
delete_entry_	34
create_link_	35
create_link_from_acl_	36
list_acl_	37
set_acl_	38
seal_object_	39
unseal_object_	41
validate_access_	42
create_segment_	43
truncate_segment_	44
set_max_length_	45
create_special_segment_	46
map_segment_	47
segment_address_	48
segment_entry_	49
unmap_segment_	50
list_mapped_segments_	51
wire_	52
unwire_	53
prepage_storage_	54
postpurge_storage_	55
update_storage_	56
deactivate_	57
move_data_	58
invoke_domain_	59
create_task_	63
get_task_meters_	64
get_task_state_	65

set_task_state_	66
get_task_scheduling_parameters_	67
set_task_scheduling_parameters_	68
activate_task_	69
deactivate_task_	70
suspend_task_	71
start_task_	72
kill_task_	73
monitor_task_	74
wakeup_	75
block_	76
set_wakeup_mask_	77
declare_broadcast_event_	78
broadcast_	79
delete_broadcast_event_	80
set_alarm_	81
set_cpu_alarm_	82
set_trap_handlers_	83
get_trap_handlers_	84
add_terminal	85
attach_terminal_	86
detach_terminal_	87
order_terminal_	88
terminal_read_	89
terminal_read_char_	90
terminal_input_available_	91
terminal_write_	92
create_channel_	93
connect_	94
disconnect_	95
identify_caller	96
send_	97
receive_	98
receive_packet_info_	99
flush_channel_	100
create_dummy_object_	101
set_switch_	102
object_status_	103
put_property	105
get_property	106

put_property_test_	107
delete_property_	108
delete_property_test_	109
test_property_	110
list_properties_	111
set_default_volume_	112
set_default_volume_domains_	113
create_quota_account_	114
delete_quota_account_	115
set_quota_limit_	116
list_quota_	117
salvage_domain_	119
priv_delete_	120
adopt_segment_	121
sweep_partition_	122
unique_id_	123
clock_read_	124
clock_set_	125
add_processor_	126
delete_processor_	127
add_memory_	128
delete_memory_	129
shutdown_	130
mount_volume_	131
dismount_volume_	132
mount_partition_	133
dismount_partition_	134
list_volumes_	135
list_partitions_	136
salvage_volume_	137
salvage_partition_	138
format_volume_	139
format_partition_	140
delete_partition_	141
read_vtoce_	142
read_vsmte_	143
delete_vtoce_	144
get_kernel_address_	145
copy_out_	146
copy_syserr_log_	147

